

INFORMATIQUE MATHÉMATIQUES AUTOMATIQUE

THÈSE DE SPÉCIALITÉ
INFORMATIQUE

CONTRIBUTION À UNE DESCRIPTION
SYNTAXIQUE ET SÉMANTIQUE D'UN
LANGAGE EXTENSIBLE

UNIVERSITÉ
DE
NICE

MARS 1976

Guy PERRIN



iman

THÈSE

présentée

à L'UNIVERSITÉ DE NICE

pour obtenir le grade de
DOCTEUR DE SPÉCIALITÉ
"INFORMATIQUE"

par

Guy PERRIN

CONTRIBUTION A UNE DESCRIPTION SYNTAXIQUE ET SEMANTIQUE
D'UN LANGAGE EXTENSIBLE

Soutenue le 5 Mars 1976, devant la Commission d'Examen :

MM. J.C. BOUSSARD, président

J.C. MIELLOU

PH. JORRAND

O. LECARME

THÈSE

présentée

À L'UNIVERSITÉ DE NICE

pour obtenir le grade de

DOCTEUR EN SPÉCIALITÉ

"LINGUISTIQUE"

par

GUY PERRIN



CONTRIBUTION À UNE DESCRIPTION SYNTAXIQUE ET SEMANTIQUE

D'UN LANGAGE EXTENSIBLE

Soutenue le 2 Mars 1972, devant la Commission d'Examen :

M. J.-C. BOUSSEARD, Président

J.-C. NIELLOU

PH. JORRARD

G. LECARME

Alain Haurat m'a proposé que nous réfléchissions à un modèle de définition de langages : je l'en remercie puis que depuis plus de deux ans que nous travaillons ensemble sur ce sujet, au laboratoire de Calcul de l'Université de Bourgogne, nos discussions, qui sont souvent passionnées, sont également fructueuses : cette thèse le prouve, qui expose une petite partie de leurs résultats.

Je suis heureux de remercier Marcien J. C. Bouslard, qui a accepté de patronner ce travail, et grâce à qui nos séjours au laboratoire d'Informatique de Nice furent, et sont toujours, aussi studieux qu'agréables. Son intérêt pour notre travail, confirmé par l'honneur qu'il me fait de présider ce jury, nous est un précieux encouragement.

Philippe Torrand a accepté de faire partie de mon jury, montrant ainsi son intérêt pour ce travail, que nous lui avons exposé à Grenoble : je

lui en suis très reconnaissant.

Je remercie Olivier Lecarme de sa présence dans ce jury.

Je tiens particulièrement à exprimer ma gratitude envers Monsieur J.C. Millon, qui pendant les quatre années passées dans son laboratoire, m'a toujours encouragé et aidé à travailler en informatique, et qui a, très sympathiquement, consacré son intérêt pour notre travail. J'ai pour ces raisons souhaité qu'il soit présent dans ce jury et le remercie vivement d'avoir accepté.

Enfin, je dois la réalisation matérielle de cette thèse à Mesdames Forster et Bourgeois, ainsi qu'à Monsieur Viana : je les remercie de la qualité de leur travail.

TABLE DES MATIERES

INTRODUCTION	1
CHAPITRE 1 - DESCRIPTION SEMANTIQUE FORMELLE	4
1.1 - Les méthodes non interprétatives	6
1.1.1 - Les méthodes axiomatiques	6
1.1.2 - Les méthodes fonctionnelles	8
1.2 - Les méthodes interprétatives	9
1.2.1 - La description formelle d'ALGØL 68	9
1.2.2 - La description sémantique dans le formalisme de Vienne	24
1.2.3 - La méthode de description par attributs sémantiques	34
CHAPITRE 2 - NOTION DE LANGAGE EXTENSIBLE	44
2.1 - Définition de l'extensibilité	45
2.1.1 - Introduction	45
2.1.2 - Principes de l'extensibilité	46
2.1.3 - Types d'extensibilité introduite par les méthodes interprétatives	46
2.2 - Extensibilité syntaxique par macros	48
2.2.1 - Principe	48
2.2.2 - Exemple et conséquences	49
2.2.3 - Restrictions au procédé	50
2.3 - Exemples d'extensibilité de la sémantique statique	52
2.3.1 - Les types de valeurs dans le système ECL	52
2.3.2 - Les classes dans SIMULA 67	53
2.3.3 - Les déclarations de mode ALGØL 68	54

CHAPITRE 3 - MODELE DE FORMALISATION	58
3.1 - Les actions primitives	60
3.1.1 - L'allocation	61
3.1.2 - L'écriture	61
3.1.3 - La lecture	63
3.1.4 - La désignation	63
3.2 - Les différentes classes d'objets	65
3.2.1 - Constante de valeur	65
3.2.2 - Constante d'adresse	65
3.2.3 - Constante de descripteur	66
3.2.4 - Constante de format	66
3.2.5 - Constante de notion	66
3.3 - Les grammaires conceptuelle et opérationnelle	67
3.3.1 - La grammaire conceptuelle	67
3.3.2 - La grammaire opérationnelle	68
3.3.3 - L'intervention des grammaires	69
3.4 - Les valeurs externes	70
CHAPITRE 4 - LE LANGAGE DE BASE	72
4.1 - La grammaire conceptuelle	75
4.2 - Exemples	77
4.2.1 - Les constantes simples	78
4.2.2 - Les constantes composées	82
4.2.3 - Les constantes d'adresse simple	85
4.2.4 - Les constantes d'adresse composée	91
4.3 - La grammaire opérationnelle	96
4.3.1 - L'opérateur "déclaration"	97
4.3.2 - L'opérateur "effectivation"	98

CHAPITRE 5 - LE PROCESSUS D'EXTENSIBILITE	100
5.1 - Arbre d'extensibilité	102
5.1.1 - Niveau d'extensibilité	102
5.1.2 - Direction d'extensibilité	103
5.1.3 - Arbre d'extensibilité	103
5.2 - Programme d'extensibilité	104
5.2.1 - Le primitif "notion"	105
5.2.2 - Programme d'extensibilité	108
5.3 - Extensibilité ALGØL 68	111
5.3.1 - Problèmes posés par cette extension	111
5.3.2 - Programme d'extensibilité	112
5.3.3 - Le nouvel axiome de la grammaire	116
5.4 - Mise en oeuvre de l'extensibilité	117
CONCLUSION	121
BIBLIOGRAPHIE	123

INTRODUCTION

Le travail exposé ici est la première réalisation d'un projet dont le but est de proposer un nouveau formalisme de description des langages de programmation évolués.

Nous nous sommes attachés à deux principes essentiels dans l'élaboration de ce formalisme : une formalisation de la sémantique des langages et une définition extensible de ces langages.

Il semble que, pour ce qui concerne les langages de programmation, les problèmes touchant aux questions de **description** syntaxique ont été abordés sous des angles très divers et ont donné lieu à des développements insoupçonnés : par contre l'étude n'apparaît encore que très parcellaire pour ce qui concerne les problèmes posés par la sémantique et sa description formelle, même si l'intérêt des solutions déjà proposées est d'ores et déjà visible dans des méthodes interprétatives de description de compilateurs.

Outre l'intérêt que peut présenter en soi une formalisation de la sémantique, nous nous attacherons à proposer une formalisation sémantique qui soit appropriée au processus d'extensibilité tel que nous le définissons.

La recherche d'une définition extensible des langages, dont le but est de permettre à chaque utilisateur d'adapter le langage, dans lequel il programme, à une classe particulière de problèmes, n'a jusqu'à présent pas connu de développements suffisants qui rendent son application réellement efficace. La cause en est sans doute qu'on a jusqu'alors envisagé la mise en oeuvre de l'extensibilité comme la définition d'un mécanisme très particulier, greffé sur une description formelle classique des langages.

Aussi peut-on penser qu'une description formelle d'un langage de programmation posant comme premier le principe de l'extensibilité est apte à démontrer l'efficacité d'une telle conception ; nous voulons dire par là que si le processus d'extensibilité est tel qu'il est impossible de discerner ce

qui a été créé par extensibilité de ce qui faisait partie, initialement, du langage, alors le résultat d'une extension cesse d'être remarquable ou exceptionnel : le processus d'extensibilité devient lui-même mode de description du langage engendré.

Dans le chapitre 1, nous rappelons les méthodes classiques de formalisation sémantique, répertoriées en 3 catégories :

- les méthodes axiomatiques,
- les méthodes fonctionnelles,
- les méthodes interprétatives.

Nous exposons brièvement le principe des méthodes non interprétatives qui, à notre connaissance, n'ont pas été appliquées efficacement à la définition de langages. Pour les méthodes interprétatives, maintenant couramment utilisées dans la définition de langages, nous exposons 3 exemples :

- le premier, est l'introduction de la sémantique dans le rapport de définition du langage ALGØL 68.
- le deuxième est connu sous le nom de formalisme de Vienne,
- le troisième est la méthode de description par attributs sémantiques.

Sur ces méthodes interprétatives sont greffés différents mécanismes d'extensibilité qui concernent respectivement : l'extension de la syntaxe, l'extension de la sémantique statique, enfin l'extension de la sémantique dynamique.

Nous présentons dans le chapitre 2 ces mécanismes d'extensibilité en montrant les principes et les difficultés :

- d'une part, de l'extensibilité de la syntaxe, réalisée par la déclaration de macros syntaxiques, définissant de nouvelles règles de la grammaire,
- d'autre part, de l'extensibilité de la sémantique statique, étudiée dans les langages EL1, SIMULA 67 et ALGOL 68.

En conclusion, nous posons le problème de l'extension de la sémantique dynamique, tel qu'il n'est pas, à notre connaissance, résolu.

Au chapitre 3 nous proposons un modèle de définition des langages, incluant une formalisation de la sémantique, tel que cette définition soit extensible : cette formalisation de la sémantique est réalisée par la définition de classes d'objets engendrées par une grammaire, dite conceptuelle. Les traitements sur les objets du langage sont produits par une grammaire, dite opérationnelle.

Ce modèle de description est davantage développé dans le chapitre 4, où il est appliqué à la définition du langage de base, c'est-à-dire le langage retenu pour servir d'initialisation au processus d'extensibilité. Les seuls objets du langage étudiés dans ce travail sont de type valeur : les classes qu'ils constituent sont engendrées par la grammaire conceptuelle dont nous donnons les règles, et que nous illustrons par des exemples. Nous présentons de manière informelle la grammaire opérationnelle du langage de base et nous exposons le rôle et les relations de ces deux grammaires, dans la définition et la reconnaissance du langage de base.

Nous montrons, au chapitre 5, comment le modèle proposé au chapitre 3, permet une définition extensible des langages, par la définition, la reconnaissance et l'accès à de nouvelles classes d'objets. La mise en oeuvre de cette propriété d'extensibilité de la définition, par un processus unique, répond en particulier au problème, posé au chapitre 2, de l'extension de la sémantique dynamique. Nous montrons, à titre d'exemple, comment le processus d'extensibilité permet d'engendrer, à partir des classes de valeurs du langage de base défini au chapitre 4, des classes de valeurs du langage ALGOL 68.

CHAPITRE 1

DESCRIPTION SEMANTIQUE FORMELLE

CHAPITRE 1

DESCRIPTION SEMANTIQUE FORMELLE

INTRODUCTION

La définition et l'utilisation de tout langage, et en particulier d'un langage de programmation, comporte deux aspects : l'aspect syntaxique et l'aspect sémantique.

La définition syntaxique des langages de programmation évolués est formalisée par un ensemble de règles de grammaire engendrant un ensemble de phrases, construites à partir d'un ensemble donné de symboles terminaux : une telle définition permet la reconnaissance automatique des phrases du langage, selon un algorithme déterminant l'évolution de l'état d'une machine abstraite (automate à nombre fini d'états, ou automate à pile, par exemple).

La définition d'une sémantique pour un langage est l'association d'une signification à toute phrase de ce langage : pour un langage de programmation la signification d'une phrase, ou valeur sémantique, est sensée être exprimée, si le compilateur du langage considéré remplit correctement ses fonctions, par la séquence d'instructions en code machine engendrée pour cette phrase par le dit compilateur. L'implémentation correcte d'un langage de programmation sous entend donc la définition d'une sémantique qui contienne toutes les indications nécessaires à l'interprétation de ses phrases. Il faut, de plus, que cette définition soit suffisamment formelle si l'on veut réaliser une implémentation de ce langage sur une classe de machines analogues.

Enfin, si la description de la sémantique d'un langage est nécessaire à sa définition, c'est-à-dire à son implémentation, il est également indispensable pour l'utilisation de ce langage de posséder la valeur sémantique de toutes les phrases écrites ; une description formelle de la sémantique d'un langage doit donc permettre une utilisation efficace du langage par un programmeur.

Cette efficacité devrait notamment se manifester lors de problèmes de terminaison ou de correction de programme, en fournissant des méthodes de preuves pour les langages évolués.

Il semble que ces différents objectifs que peuvent se fixer des concepteurs de modèles de formalisation sémantique aient été jusqu'alors difficilement conciliables sinon contradictoires : le nombre d'approches différentes de la sémantique récemment proposées s'explique en partie par la diversité des buts poursuivis.

Nous retiendrons la classification proposée par DIJKSTRA dans [Li 72] pour ces différentes méthodes, et nous examinerons respectivement :
les méthodes dites axiomatiques,
les méthodes dites fonctionnelles,
enfin les méthodes dites interprétatives.

Alors que les méthodes interprétatives définissent la valeur sémantique d'un couple (programme, données), en définissant un "calcul"(*) délivrant un "résultat", les autres méthodes définissent la valeur sémantique associée à un programme, comme une application de l'ensemble des données dans l'ensemble des calculs.

1.1. LES METHODES NON INTERPRETATIVES

1.1.1. - Les méthodes axiomatiques

Les méthodes, dont le principe fut proposé par HOARE [Ho 69] reposent sur la définition d'axiomes permettant d'établir, sous forme de prédicats logiques, les propriétés des objets et constructions d'un langage. La sémantique d'un programme est alors établie, à partir des axiomes relatifs aux données, au moyen d'un ensemble de règles de déduction, qui définissent le théorème relatif à ce programme.

Si on note $P, Q, R, \dots$, l'ensemble des prédicats logiques décrivant l'ensemble des propriétés des données du langage, on peut associer à chaque instruction S du langage, un théorème du type :

(*) Nous employons le terme "calcul" tel qu'il est défini dans [Pa 73], comme une suite de fonctions élémentaires sur une structure d'informations, cette définition recouvrant la notion intuitive de calcul d'un ordinateur ou de calcul d'un automate.

$$H = P \{ S \} Q$$

qui signifie que si le prédicat P est vrai, avant l'exécution de S , le prédicat Q est vrai après son exécution.

Les règles de déduction entre théorèmes permettent de définir le théorème relatif à un programme : on peut donner pour exemple de règles de déduction :

$$\bullet \frac{H_1, H_2, \dots, H_n}{H}$$

qui signifie : si $H_1, H_2, \dots$ et H_n sont des théorèmes, alors H est un théorème.

$$\bullet \frac{H_1, H_2, \dots, H_n \vdash H_{n+1}}{H}$$

qui signifie : si les théorèmes $H_1, H_2, \dots, H_n$ impliquent le théorème H_{n+1} , alors H est un théorème.

Exemple : (d'après [Ho W 72])

On considère un langage dans lequel, tout programme PR est constitué d'un ensemble D de déclarations, suivi d'une liste LI d'instructions :

La règle de déduction associée à une liste d'instructions, et exprimant leur séquençement à l'aide du point-virgule, est la suivante :

$$\bullet \frac{P \{ LI \} Q, Q \{ S \} R}{P \{ LI ; S \} R}$$

La règle de déduction associée à un programme est :

$$\frac{H \vdash P \{ LI \} Q}{P \{ D ; LI \} Q}$$

où H est l'ensemble des prédicats logiques décrivant les propriétés des objets déclarés dans D .

Remarques :

1 - Ces méthodes, en fait purement logiques, qui établissent "statiquement" les propriétés d'un programme, sans exécution effective du calcul de ce programme, ne peuvent que traduire assez mal des contraintes propres à un langage de programmation.

D'ailleurs, à notre connaissance, les seuls essais de définition de langages existants, par des méthodes axiomatiques n'ont pas mené à des résultats satisfaisants : ainsi, il semble que dans la définition de PASCAL [Ho W 72] , on soit amené à ignorer bien des contraintes du langage, dont la plus évidente est l'instruction de rupture de séquence.

2 - En outre, parce que ces méthodes s'organisent autour de concepts très mathématiques, il paraît difficile de déduire automatiquement de sa définition, un modèle d'implémentation pour un langage ainsi décrit.

3 - Cependant ces méthodes semblent appelées à présenter de grands développements dans la résolution de problèmes de preuves ou de corrections de programmes.

1.1.2. - Les méthodes fonctionnelles

Ces méthodes introduites par D. SCOTT [Sc S 71] consistent à considérer qu'un programme est la dénotation d'une certaine application d'un ensemble de données dans un ensemble de résultats.

Cette application traduit la valeur sémantique du programme considéré : exprimée sous forme mathématique, elle ne met en oeuvre aucun procédé algorithmique et ne peut donc pas rendre compte de la complexité du programme qui la dénote.

Ces méthodes, également, semblent intéressantes dans des problèmes de preuves de programme, mais à notre connaissance [Co 74] elles sont mal adaptées, parce que trop éloignées des réalités informatiques, à la définition d'un langage de programmation. Nous n'irons donc pas plus avant dans leur présentation.

1.2. LES METHODES INTERPRETATIVES

Les méthodes, dites interprétatives, sont actuellement les méthodes de définition de la sémantique des langages les plus répandues : nous étudierons successivement leur mise en oeuvre dans la définition du langage ALGØL 68, dans le formalisme de VIENNE, et dans la méthode de description par attributs sémantiques. On peut déjà, très grossièrement, définir le principe général de ces méthodes : l'exécution particulière d'un programme pour un certain ensemble de données définit un certain calcul mettant en oeuvre un certain nombre d'objets : à chaque pas de ce calcul, l'ensemble des informations relatives à chaque objet constitue l'état du calcul. La sémantique d'une instruction est alors définie comme une fonction de transition entre les différents états.

1.2.1. - La description formelle d'ALGØL 68 [vWM 73]

1.2.1. a. - Mode de description syntaxique

La description syntaxique d'Algol 68 est réalisée par deux niveaux de grammaires formelles :

- une première grammaire qui introduit les entités nécessaires aux contraintes contextuelles.
- une seconde que l'on peut considérer comme un ensemble de modèles de règles où doivent s'insérer les contraintes introduites par la grammaire précédente, pour définir une grammaire ayant un nombre infini de règles et qui est la grammaire du langage.

1.2.1. a. a. - La grammaire de premier niveau ou méta-grammaire :

Cette grammaire est une grammaire context-free sans définition d'axiome unique : on peut la définir comme la famille de quadruplets suivante :

$$(G (m) = \{ M , P , R , m \} \quad ; \quad m \in M)$$

où chaque quadruplet définit une grammaire context-free d'axiome m de la façon suivante :

M est un ensemble fini de symboles non terminaux, ou métanotions, notés en lettres majuscules.

P est l'ensemble des symboles terminaux, ou protonotions, notés en lettres minuscules.

R est l'ensemble des règles de métaproduction, c'est-à-dire un sous-ensemble fini de $M \times (M \cup P)^*$:

On note $X :: y_1 y_2 \dots y_n$ une telle règle,

où $X \in M$ et $y_i \in M \cup P$ pour tout i de 1 à n

On note également :

$X :: y ; z.$ si $X :: y.$ et $X :: z.$

Exemples de règles de métaproduction dans le rapport de définition Algol 68 :

NEST : : LAYER ; NEST LAYER.

LAYER : : new DECSETY LABSETY.

DEC : : MODE TAG ; priority PRIOTAD ; MOID TALLY TAB ; DUO TAD ;
MONO TAM ; MODE field TAG.

DECS : : DEC ; DECS DEC.

DECSETY : : DECS ; EMPTY.

EMPTY : : .

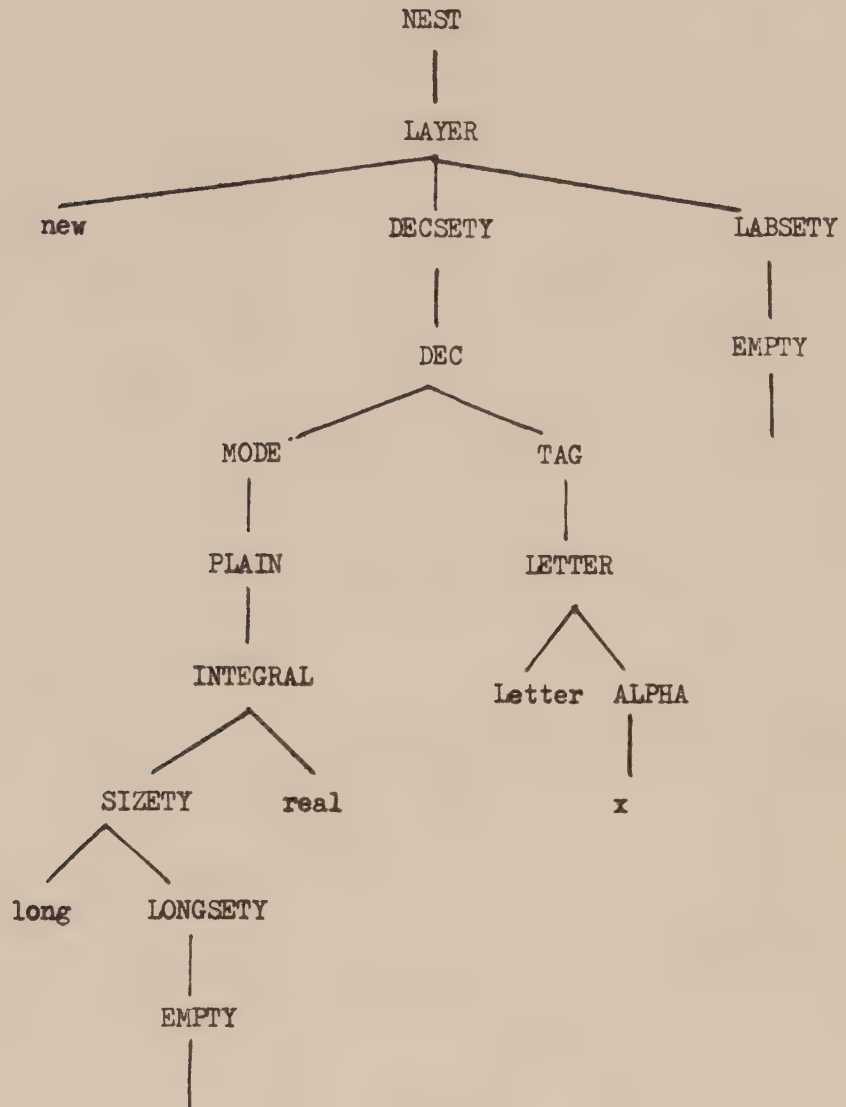
NEST, LAYER, DEC, MODE, ... , sont des métanotions.

new, priority, ... , sont des protonotions.

On appelle hypernotation un élément de $(M \cup P)^*$, c'est-à-dire, ce qui constitue généralement la partie droite d'une règle de métaproduction :

MODE field TAG, new DECSETY LABSETY, ... sont des hypernotations.

On appelle métaproduction terminale une dérivation d'une métanotation, constituée uniquement de protonotions : par exemple, "new long real letter x" est une métaproduction terminale de la métanotation NEST par la dérivation suivante :



On appelle métalangage l'ensemble infini des métaproductions terminales.

1.2.1. a. b. - La grammaire de second niveau :

C'est une grammaire définie par le quintuplet

$$G = \{ M, P, \Sigma, \sigma, H \}$$

où M est l'ensemble des métanotions de la métagrammaire

P est l'ensemble des protonotions de cette grammaire

Σ est l'alphabet du langage

σ est un élément de P , appelé axiome de la grammaire

et H est un sous ensemble fini de $(P \cup M)^+ \times ((P \cup M)^+ \cup \Sigma)^*$:

on appelle hyperrègle un élément de H .

on note $\Delta x : y_1, y_2, \dots, y_n$. une hyperrègle où

$$x \in (P \cup M)^+$$

$$\text{pour tout } i \text{ de } 1 \text{ à } n, y_i \in (P \cup M)^+$$

Exemple d'hyperrègle :

- a) Δ REF to MODE NEST assignation : REF to MODE ~~NEST~~ destination,
becomes token, MODE ~~NEST~~ source.

On déduit de chaque hyperrègle un ensemble, généralement infini, de règles de productions, qui forment la grammaire du langage, en remplaçant les métanotions qui y apparaissent par une de leur métaproduction terminale (on appelle notions, les protonotions ainsi obtenues).

Ce remplacement se fait :

- de manière uniforme, c'est-à-dire en substituant dans une même hyperrègle, à toutes les occurrences de la même métanotion, la même métaproduction terminale.

- éventuellement sous le contrôle de prédicats portant sur les métanotions.

Ce remplacement exprime ainsi dans les règles de productions les contraintes contextuelles introduites par la métagrammaire.

En remplaçant les métanotions TERTIARY et UNIT par une de leur métaproduction, il vient :

- ▲ reference to integral NEST assignation : soft reference to integral NEST applied identifier with TAG 1 coerced, becomes token, strong integral NEST applied identifier with TAG 2 coerced.

D'après les règles de métaproductions :

SOFT	:	:	unchanged from.
FORM	:	:	MORF ; COMORF.
MORF	:	:	NEST applied identifier with TAG.
STRONG	:	:	FIRM
FIRM	:	:	MEEK
MEEK	:	:	unchanged from.

et les hyperrègles :

- ▲ soft MODE FORM coerced : SOFT MODE FORM.
- ▲ unchanged from MOID FORM : MOID FORM.
- ▲ strong MOID FORM coerced : where < FORM > is < MORF > , STRONG MOID MORF.

On déduit de :

soft reference to integral NEST applied identifier with TAG 1 coerced

la phrase :

reference to integral NEST applied identifier with TAG 1.

Et on déduit de :

strong integral NEST applied identifier with TAG 2 coerced

la phrase :

integral NEST applied identifier with TAG 2.

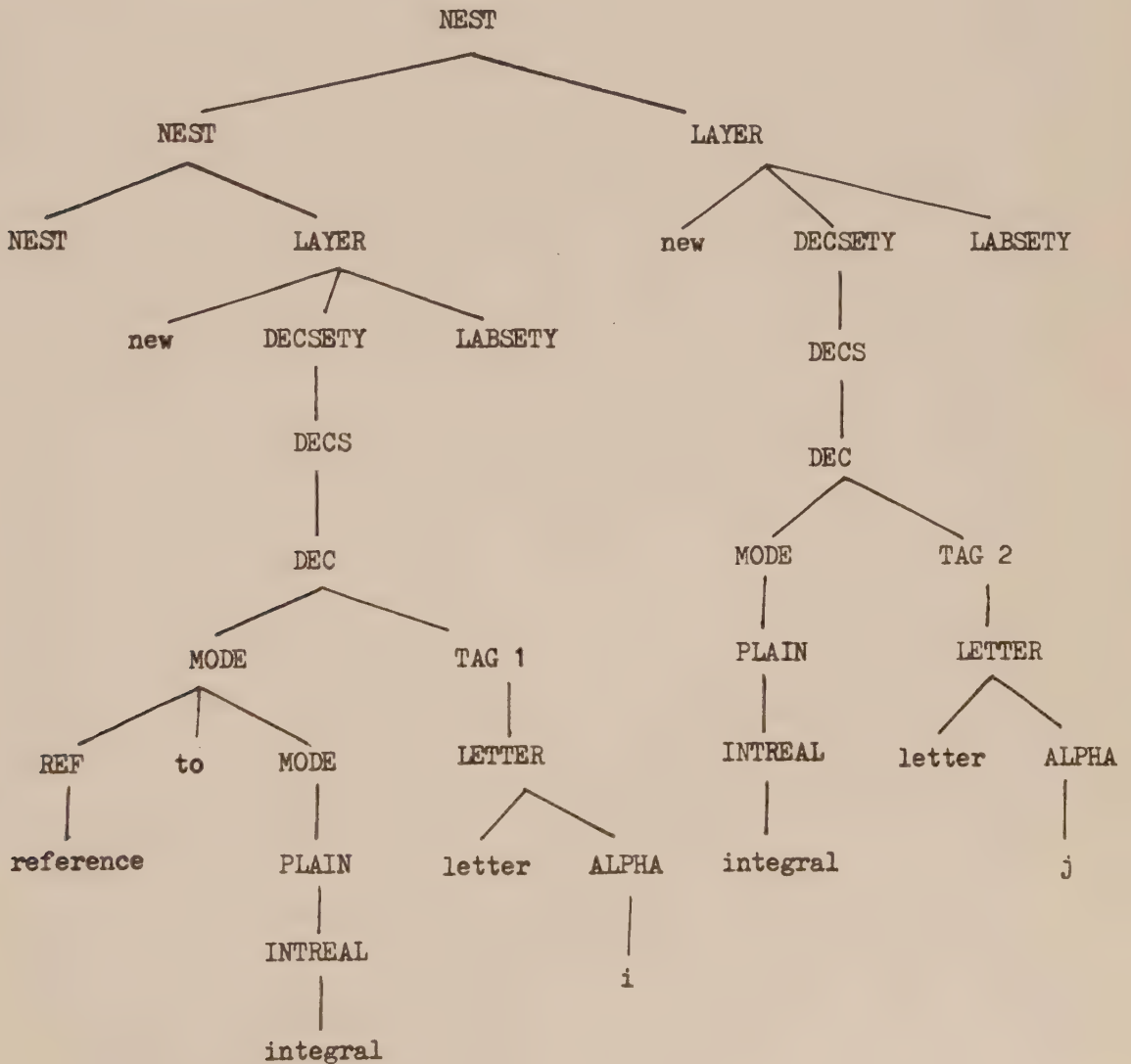
On déduit alors l'hyperrègle suivante :

- ▲ reference to integral NEST assignation : reference to integral NEST applied identifier with TAG 1, becomes token, integral NEST applied identifier with TAG 2.

La métanotion NEST apparaissant en partie gauche et aux deux membres de la partie droite de cette règle exprime les conditions de contexte d'une instruction d'affectation, conditions spécifiées par le prédicat contenu dans l'hyperrègle suivante , qui permet de poursuivre la dérivation :

▲ QUALITY NEST applied INDICATOR with TAX : where QUALITY TAX identified in NEST, TAX token.

Ce prédicat, s'il est valide, assure que les identificateurs de variables utilisés dans l'instruction d'affectation ont été déclarés, c'est-à-dire, qu'à l'instant de l'élaboration de l'affectation, la métaproduction de NEST est du type :



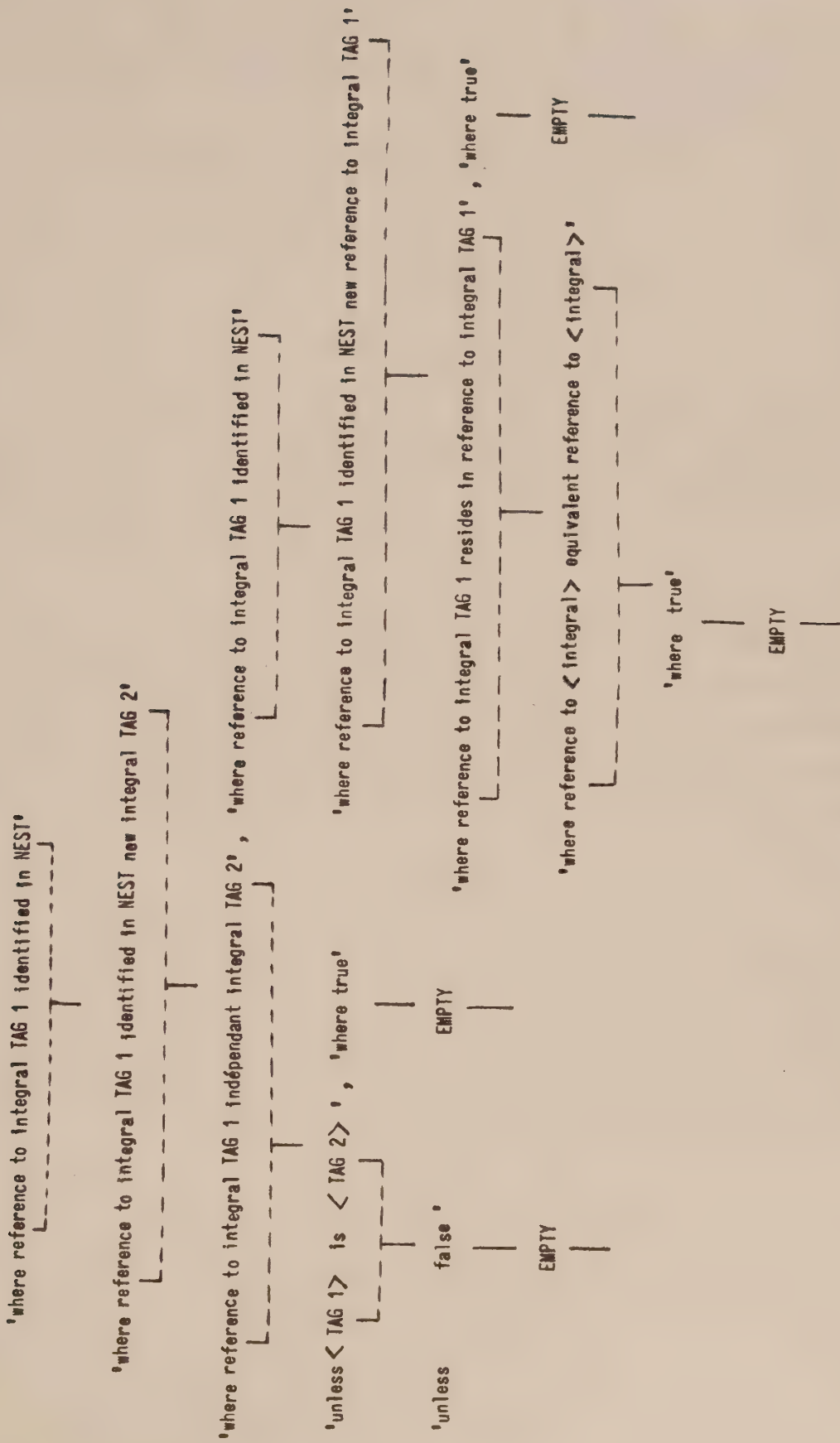
Alors les prédicats :

'where reference to integral TAG 1 identified in NEST'

et 'where integral TAG 2 identified in NEST'

sont valides.

Nous ne donnons comme exemple que la vérification de la validité du premier de ces prédicats:



La règle devient donc :

▲ reference to integral NEST assignation : TAG 1 token, becomes token,
TAG 2 token.

qui produit la phrase Algol 68 :

i := j

1.2.1. b. - Description sémantique

1.2.1. b. a. - "semantics" :

Nous venons de voir que la description syntaxique du langage Algol 68, sous forme d'hyperrègles, contient certains aspects exprimant notamment des contraintes contextuelles prises en compte pour la déduction des règles de production qui engendrent les phrases du langage.

Il faut, en outre associer une signification à toute phrase ainsi décrite, en définissant de manière algorithmique l'effet de l'élaboration de l'action qu'elle représente : ceci fait l'objet, dans le rapport de définition Algol 68, de paragraphes spéciaux faisant suite à la partie de description syntaxique correspondante : ces paragraphes sont écrits dans un sous-ensemble d'une langue naturelle auquel sont ajoutées les hypernotations de la grammaire.

1.2.1. b. b. - exemple 1 :

L'instruction d'affectation est décrite syntaxiquement à partir des trois hyperrègles a), b) et c) rappelées au paragraphe 1.2.1. a. Nous avons vu que les différentes conditions de contextes étaient prises en considération dans ces hyperrègles et comment on en déduisait une règle de production. Un paragraphe "semantics" exprime ensuite la façon dont on élabore une affectation :

"Semantics"

a) On élabore une affectation A de la manière suivante :

- Soit N et W les résultats des constituants . destination .
et . source . de A ;
- On affecte W à N ;
- Le résultat de A est N.

b) On affecte une valeur W à un nom N, dont le mode est un certain 'REF TO MODE' , de la manière suivante :

Il faut que :

- N ne soit pas nil, et que
- la portée de W ne soit pas inférieure à celle de N ;

Cas A : 'MODE' est un certain "structured with FIELDS mode" :

- pour chaque 'TAG' sélectionnant un champ dans W,
- ce champ est affecté au nom sélectionné par 'TAG' dans N ;

Cas B : 'MODE' est un certain 'ROWS of MODE 1' :

- il faut que les descripteurs de W et N soient identiques ;
- pour chaque indice I sélectionnant un élément dans W,
- cet élément est affecté au nom sélectionné par I dans N ;

Cas C : 'MODE' est un certain 'flexible ROWS of MODE 1' :

- soit V la valeur repérée par N ;
- on fait repérer par N une valeur multiple composée :
 - (i) du descripteur de W
 - (ii) de variantes d'un élément de V, s'il y en a, et sinon de n'importe quelles valeurs acceptables pour ce 'MODE 1' ;
- N est chargé de sous-noms ;
- pour tout indice I sélectionnant un élément dans W,
- cet élément est affecté au nom sélectionné par I dans N ;

Autres cas :

- On fait repérer W par N.

1.2.1. b. c. - exemple 2 ;

A la métanotion NEST apparaissant dans toute dérivation de la grammaire, les paragraphes "semantics" associent une information précisant l'environnement dans lequel doit être élaborée l'action spécifiée dans la dérivation : celui-ci est constitué des identificateurs, symboles opérateurs, indicateurs de modes, etc. valides à l'instant de l'élaboration.

Ainsi sur l'exemple précédent, l'élaboration de l'instruction d'affectation :

$i := j$

ayant pour "opérandes" des occurrences d'utilisation d'identificateurs consiste en l'exécution des actions décrites au § 1.2.1. b. b. dans un environnement résultant de la dérivation de NEST telle qu'elle est indiquée au § 1.2.1. a. c.

Cet environnement est déterminé à partir des règles suivantes, extraites d'un paragraphe "semantics" :

L'environnement E "construit"

- à partir d'un environnement E1 ■
est déterminé comme suit :
- E a une portée inférieure à E1 et est constitué de E1 et d'une nouvelle partie correspondant à 'PROPSETY' si C est présent, à 'EMPTY' sinon ;
si C est une . declarative . , une . for-part . ou une . specification . ,
et 'PROPSETY' est une certaine 'DECS' :
pour $i = 1, \dots, n$, où n est le nombre de 'DEC' s enveloppées par 'DECS',
- V est attribuée dans E au ième constituant . defining-identifier ., s'il existe, de C , et , sinon , à un . defining-indicator-with-letter-aleph . ;

1.2.1. b. d. Arbre dynamique :

Un programme Algol 68 est une phrase du langage dérivant de la notion : program, par la règle de production :

▲ program : strong void new claused clause .

qui définit l'élaboration d'un programme comme l'élaboration d'un ensemble d'actions dans un environnement appelé initial et constitué du prologue standard (implémentation du langage , ensemble des modes, opérateurs et fonctions standards, etc.) et éventuellement de prologues particuliers (bibliothèque, tâches système, etc.).

Cette élaboration est représentée par la construction d'un arbre, appelé arbre dynamique, dont chaque noeud correspond à l'élaboration d'une action.

- la racine de cet arbre représente l'élaboration de l'action contenue dans la notion : program.

- lorsqu'un noeud représente une action composée, c'est-à-dire constituée d'une ou plusieurs autres actions, celles-ci composent un sous-arbre descendant du noeud, et leur élaboration, qui peut être sérielle ou collatérale, est l'élaboration de l'action composée :

- . l'élaboration d'une action composée est sérielle lorsque l'activation de chaque action qui la compose a lieu lorsque l'action précédente est achevée,
- . l'élaboration d'une action composée est collatérale lorsque l'une quelconque des actions qui la composent est activée dès qu'une autre action est achevée.

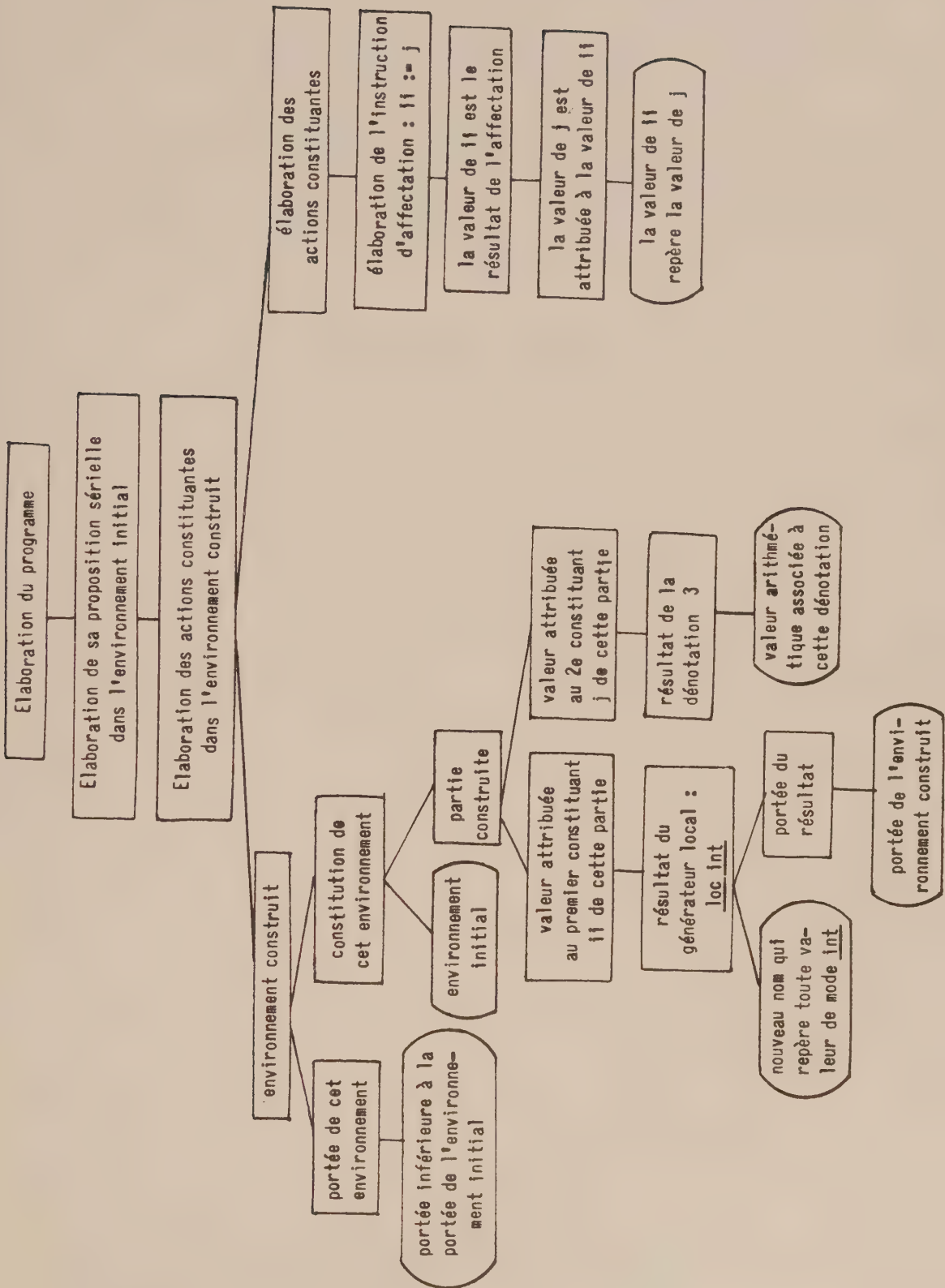
- les feuilles de l'arbre dynamique sont constituées des actions qui ne demandent pas d'élaboration, comme les symboles terminaux du langage.

L'élaboration d'une action se traduit par la suppression du noeud correspondant dans l'arbre dynamique : l'élaboration du programme est terminée lorsque l'arbre dynamique est vide.

Exemple : arbre dynamique associé au programme Algol 68 suivant :

```
begin :  
    refint  ii = loc int ;  
    int  j  = 3 ;  
    ii := j  
  
end
```

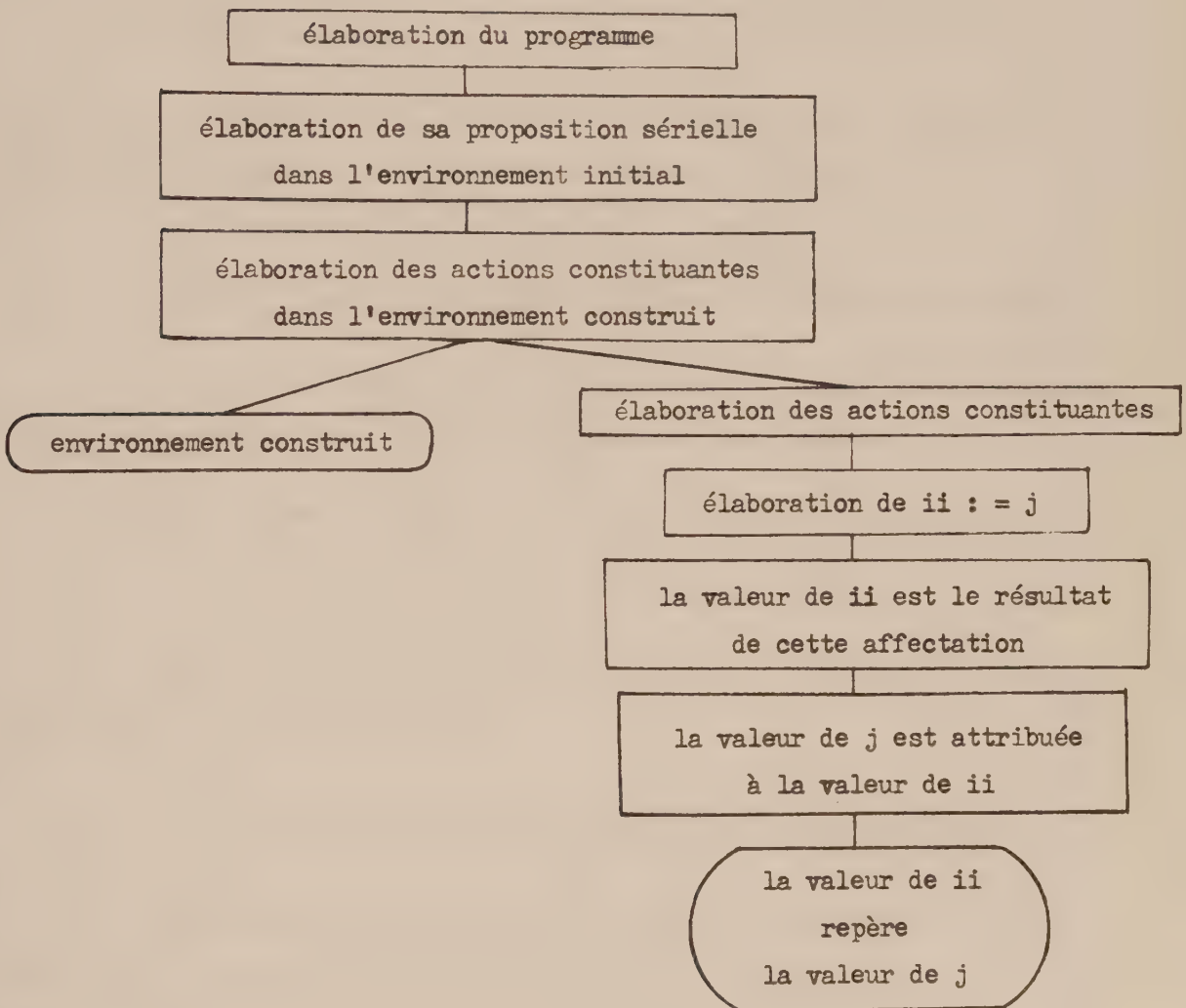
Le programme est constitué d'un ensemble d'actions dont l'exécution est sérielle : son élaboration peut se représenter par l'arbre dynamique suivant :



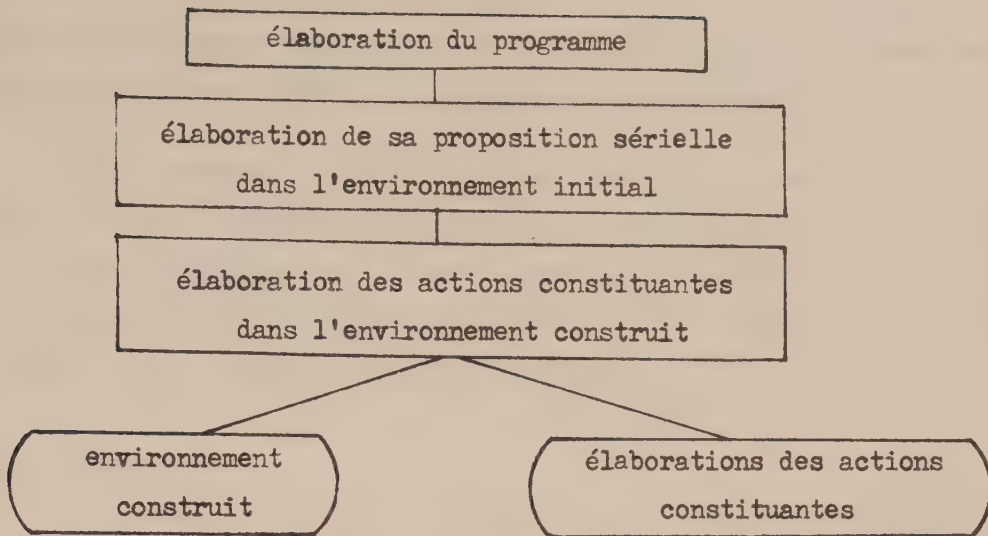
Les actions représentées dans les feuilles de cet arbre correspondent à des actions non composées qui ne demandent aucune élaboration :

par exemple : "la valeur de ii repère la valeur de j "

L'élaboration des déclarations consiste en la construction telle qu'elle est décrite au § 1.2.1. b. c. d'un nouvel environnement à partir de l'environnement initial du programme : lorsque cette élaboration est terminée, l'arbre dynamique est dans l'état suivant :



L'élaboration de l'affectation, décrite au § 1.2.1. b. b. réalise l'élaboration des actions constituantes du programme : l'arbre dynamique est devenu :



qui exprime que l'élaboration du programme est achevée par l'élaboration sérielle de ses constituants : l'arbre dynamique est alors vide.

1.2.2. - La description sémantique dans le formalisme de Vienne

Dans cette formalisation, dont le principe et les premiers exemples étaient déjà énoncée dans [Mc 66], on procède en deux temps :

- une phase de traduction du programme source au programme abstrait.
- une phase d'interprétation du programme abstrait.

L'application de ce formalisme a été faite à quelques langages, comme Algol 60 ou PL/1 [Lu W 69].

1.2.2. a. - Syntaxe concrète et langage abstrait

Le programme source est exprimé dans la syntaxe, dite concrète, syntaxe décrite par une grammaire classique : on peut distinguer dans cette expression des parties significatives et d'autres qui sont introduites, seulement pour rendre plus agréable l'utilisation du langage.

A partir de cette syntaxe concrète, on exprime, dans une première phase, le programme sous forme de "programme abstrait" en ne conservant que les parties significatives, qui sont nécessaires à l'interprétation sémantique :

l'intérêt de cette présentation est de séparer les problèmes de représentation et de notation, des problèmes d'interprétation sémantique.

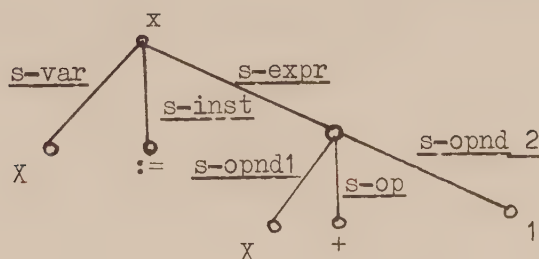
Les "programmes abstraits" constituent le langage, dit langage abstrait : ils sont définis comme des objets composés à partir d'objets élémentaires appelés branches et désignés chacun par un sélecteur : ce sélecteur sert à la fois d'identificateur pour la branche et de fonction permettant de sélectionner l'objet accroché à cette branche.

Exemple : [Ne 71]

Une instruction d'affectation, exprimée dans la syntaxe concrète par

$X := X + 1$

peut se représenter par l'objet x du langage abstrait, objet composé de trois branches désignées respectivement par s-var, s-inst et s-expr.



Ces identificateurs de branches permettent de sélectionner les objets qui constituent l'objet x .

La sélection s-expr (x) détermine la source de cette affectation, source décrite par un objet du langage abstrait constitué de trois branches, respectivement sélectionnées par :

s - opnd 1 (s - expr (x)) qui détermine la variable X

s - op (s - expr (x)) qui détermine le signe opératoire +

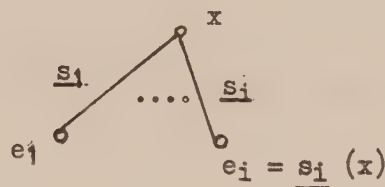
et s - opnd 2 (s - expr (x)) qui détermine la constante 1.

1.2.2. b. - Opérateurs sur les objets du langage abstrait

[Lu W 69] et [Ar G 74]

La construction des objets du langage abstrait est formalisée par la définition des trois opérateurs μ_o , μ et δ qui réalisent respectivement, la création, le branchement et la suppression d'objets. Ces opérateurs sont définis de la façon suivante :

On désigne par x un objet du langage abstrait, dont on peut sélectionner un constituant e_i par le sélecteur s_i :



On appelle ensemble caractéristique $\overline{x}$ de l'objet x, l'ensemble des paires $\langle \underline{s} : e \rangle$ telles que $\underline{s}(x) = e$:

$$\overline{x} = \{ \langle \underline{s} : e \rangle \mid \underline{s}(x) = e \}$$

Nous verrons au paragraphe suivant que, pratiquement on définit un objet par son ensemble caractéristique.

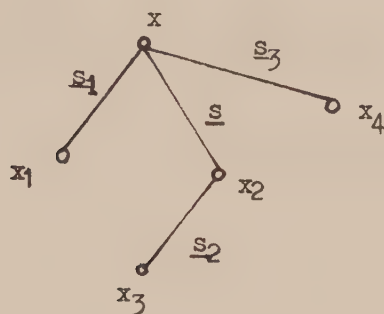
L'opérateur μ_o réalise la création d'un objet dont l'ensemble caractéristique est :

$$\overline{\mu_o(\underline{s} : x)} = \{ \langle \underline{s} : x \rangle \}$$

L'opérateur μ , opérateur de branchement, est défini par :

$\mu(x ; \langle \underline{s} : y \rangle)$ est l'objet obtenu en substituant l'objet y à l'objet sélectionné par $\underline{s}$ dans x .

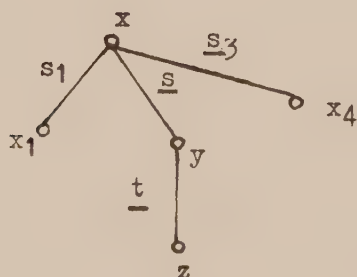
Exemple : si x est l'objet suivant :



et si y est l'objet suivant :



$\mu(x ; \langle \underline{s} : y \rangle)$ est l'objet suivant :



d'une façon générale, l'ensemble caractéristique de l'objet

$\mu(x ; \langle \underline{s} : y \rangle)$ est défini par :

$$\overline{\mu(x; \langle \underline{s} : y \rangle)} = \{ \langle \underline{\sigma} : e \rangle \mid \langle \underline{\sigma} : e \rangle \in \overline{x} \text{ et } \underline{\sigma} \notin \text{Sog} \} \cup \\ \{ \langle \underline{\sigma} \circ s : e \rangle \mid \langle \underline{\sigma} : e \rangle \in \overline{y} \}$$

où S est l'ensemble des sélecteurs de l'objet x.

L'opérateur δ , défini par :

$\delta(x; \underline{s}) = \mu(x; \langle \underline{s} : \Omega \rangle)$, où Ω désigne l'objet vide, réalise la suppression de l'objet constituant de x, sélectionné par $\underline{s}$.

1.2.2. c. - Syntaxe abstraite

L'ensemble des objets du langage abstrait est défini par une syntaxe, dite syntaxe abstraite, exprimée sous la forme de prédicats logiques qui testent la nature de chaque objet par l'intermédiaire de ses constituants obtenus par sélection.

Exemple : définition partielle de la syntaxe abstraite d'un langage où tout programme est constitué d'un ensemble de déclarations, suivi d'une liste d'instructions.

Les prédicats sont précédés du préfixe **is**,
les sélecteurs sont précédés du préfixe **s-**,
les objets sont représentés par leur ensemble caractéristique :

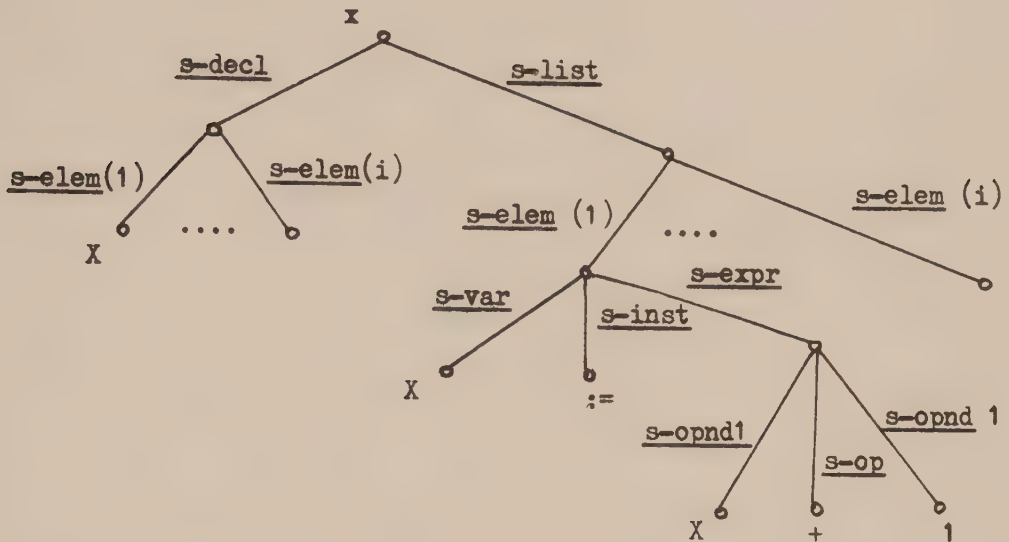
is - programme = $\langle \underline{s-decl} : \text{is - ident - liste} \rangle$; $\langle \underline{s-list} : \text{is-inst-liste} \rangle$

Le prédicat **is-programme** est valide pour les objets constitués de 2 branches :

- l'une désignée par s-decl, permettant de sélectionner un objet pour lequel le prédicat **is-ident-liste** est valide,
- l'autre désignée par s-list, permettant de sélectionner un objet pour lequel le prédicat **is-inst-liste** est valide.

Nous ne donnons pas en détail dans cet exemple, la définition d'un objet de type liste : nous rappelons simplement qu'un objet x, pour lequel le prédicat **is-truc-liste** est valide est un objet constitué d'objets pour lesquels le prédicat **is-truc** est valide, chacun de ses objets étant sélectionné par un sélecteur s-elem (i) appliqué à l'objet x.

Exemple de programme abstrait, respectant cette syntaxe :



1.2.2. d. - Interpréteur abstrait

La première phase a consisté à exprimer le programme source sous forme abstraite : la seconde consiste à interpréter le programme abstrait obtenu : ceci est formalisé par la définition d'un interpréteur abstrait.

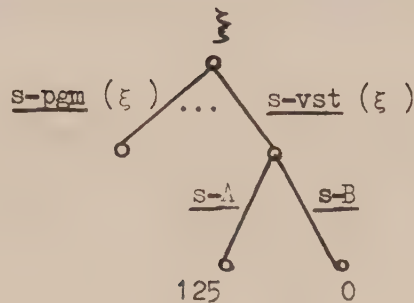
1.2.2. d. a. - définition :

Un interpréteur abstrait est une machine abstraite comportant cinq paramètres dont les valeurs déterminent l'état ξ de la machine. Ces paramètres sont :

- 1 - le programme abstrait, sélectionné par s-pgm (ξ)
- 2 - un compteur d'instructions dont la valeur indique le rang dans le programme considéré de l'instruction interprétée.

3 - l'ensemble des valeurs, sélectionné par s-vst (ξ), dont on sélectionne chaque élément par l'identificateur de la variable correspondante :

Exemple :



4 - le répertoire de l'interpréteur, comprenant les instructions de l'interpréteur : instructions de base ou macro-instructions.

5 - l'arbre de contrôle, dont chaque noeud est une instruction faisant partie du répertoire de l'interpréteur, et dont nous verrons la construction au § 1.2.2. d. b.

La description sémantique consiste en la définition d'une fonction de transitions pour cette machine abstraite : chaque application de cette fonction de transition se traduit par l'évolution d'un des paramètres caractérisant l'état de la machine.

1.2.2. d. b. - l'arbre de contrôle :

L'arbre de contrôle dans le formalisme de Vienne a un rôle identique à l'arbre dynamique tel que sa construction est suggérée dans le rapport de définition du langage ALGØL 68 (cf. § 1.2.1. b. c.).

Il est initialisé par une macro-instruction qui assure l'enchaînement des instructions constituant le programme abstrait, l'algorithme assurant cet enchaînement procédant à chaque étape, à une succession de tests sur la validité de prédicats portant sur la nature de l'instruction ou de l'expression considérée.

L'interprétation d'une instruction du programme, correspondant à un noeud de l'arbre de contrôle, est réalisée par le sous arbre accroché à ce noeud : l'élaboration de cette instruction est l'élaboration de toutes les instructions exprimées dans ce sous arbre, à partir des feuilles, qui indiquent l'accès aux valeurs par une instruction du répertoire de l'interpréteur.

L'évaluation de sous-expressions, ou l'élaboration d'instructions exprimées dans un sous-arbre accroché à un noeud, produit des résultats intermédiaires, à utiliser postérieurement dans l'exécution de l'instruction associée au noeud considéré. La conservation de ces résultats intermédiaires et leur utilisation ultérieure est réalisée dans l'interpréteur abstrait par le procédé dit du retour d'information : le sous-arbre considéré est alors vide.

Exemple :

Nous montrons sur l'exemple simple du § 1.2.2. a. comment évolue l'arbre de contrôle associé à l'instruction d'affectation au cours de l'interprétation de cette instruction.

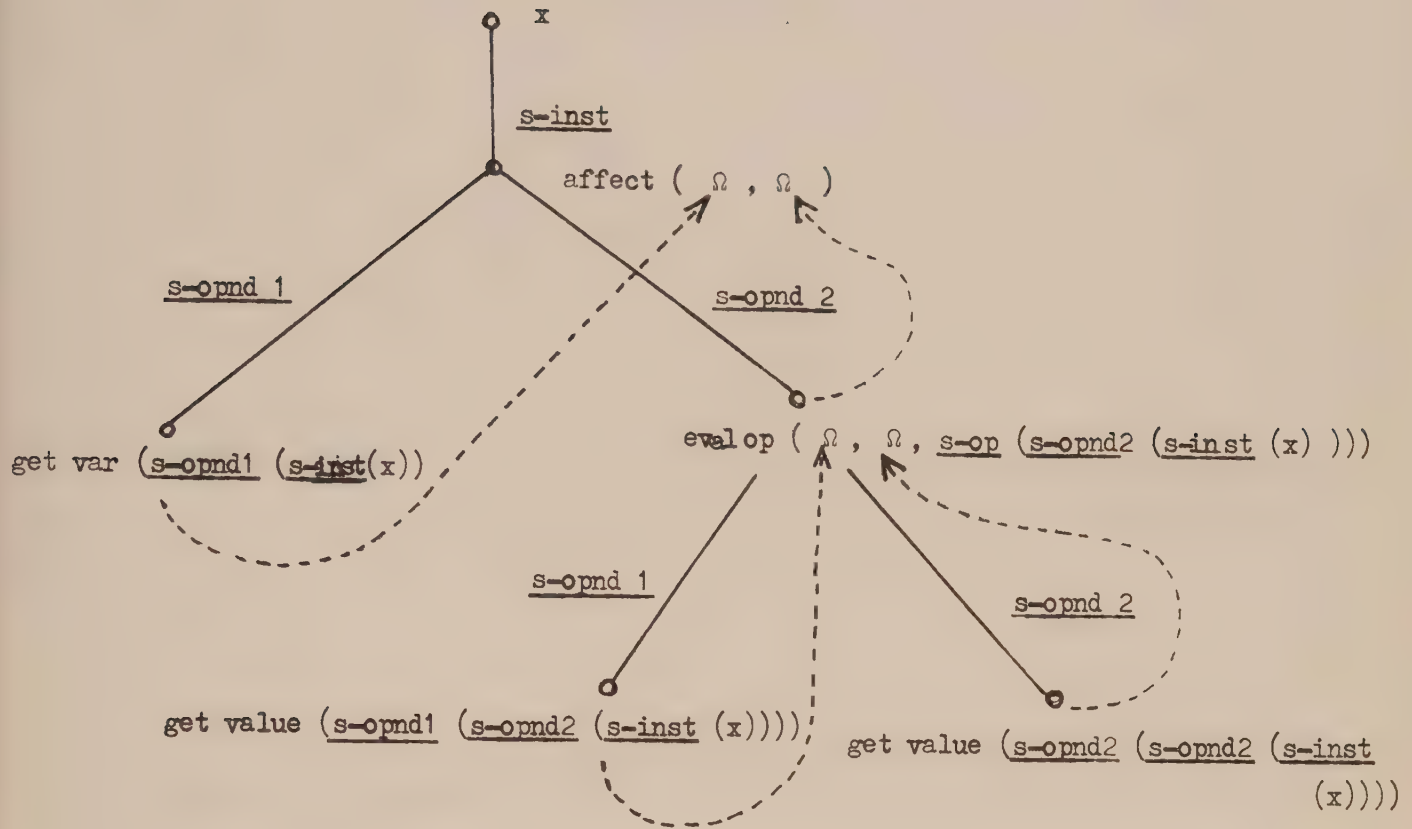
Les instructions du répertoire de l'interpréteur, utilisées dans cet exemple, sont :

affect (,) : instruction d'affectation à deux paramètres.

evalop (, ,) : instruction d'évaluation d'une expression, dont les paramètres sont les opérandes de cette expression et l'opérateur di-adique.

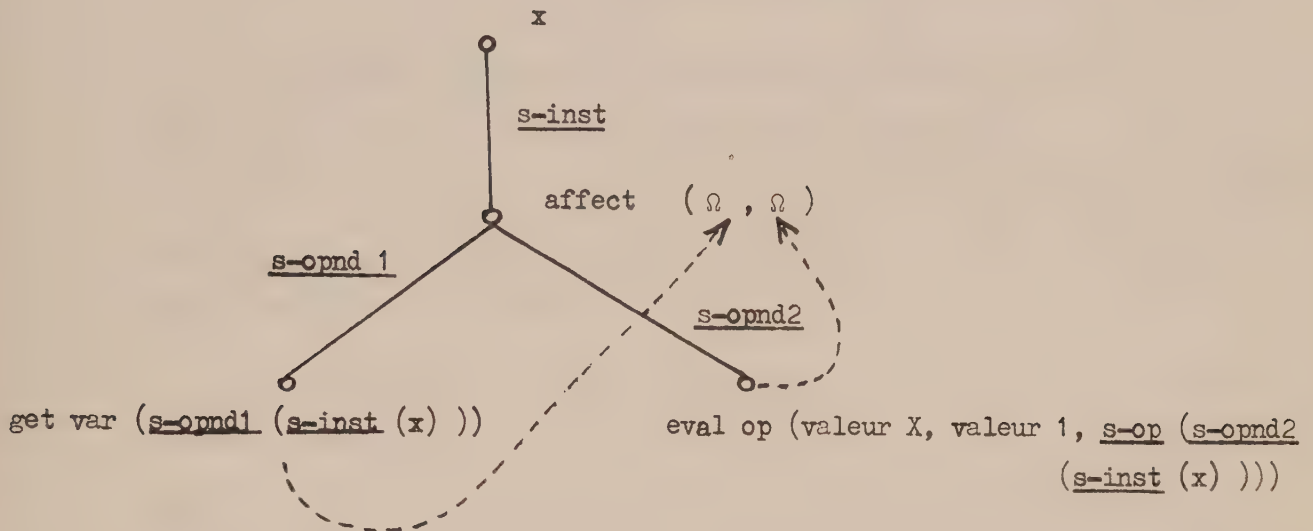
get var () et get value () : instructions à un paramètre réalisant l'accès aux valeurs.

Nous indiquons par des flèches en pointillé, la position à laquelle doit s'insérer, par retour d'information, toute valeur intermédiaire au cours de l'élaboration.



_ Les résultats de l'élaboration des instructions `get value`, sont les valeurs de la variable `X` et la valeur de la constante 1.

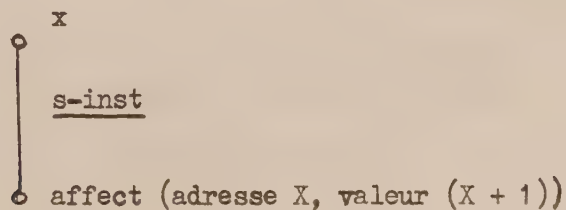
Le retour d'information se traduit par la nouvelle valeur de l'arbre de contrôle :



- Le résultat de l'élaboration de l'instruction `eval op`, est la valeur de l'expression $X + 1$.

- Le résultat de l'élaboration de l'instruction `get var` est l'adresse de la variable identifiée par X .

- Après retour d'information on obtient pour nouvelle valeur de l'arbre de contrôle :



Lorsque l'arbre de contrôle est vide, l'interprétation est terminée.

1.2.3. - La méthode de description par attributs sémantiques
[Kn 71], [Lo 74] et [Lo 75].

1.2.3. a. - Attributs synthétisés et attributs hérités :

La méthode de description de langage par attributs sémantiques déjà utilisée pour l'écriture de compilateurs (projet DELTA [Lo B 73] et compilateur Algol 68 [Cu D 75]) consiste à définir pour tout symbole, terminal ou non, d'une grammaire context-free, un ensemble d'attributs qui permettent l'interprétation sémantique des productions terminales dans la dérivation desquelles apparaît ce symbole.

On distingue deux types d'attributs :

- les attributs synthétisés, qui définissent les propriétés attachées au symbole considéré, qui sont fonction seulement des attributs de ses dérivations.

- on appelle hérités, les attributs qui ne sont pas synthétisés : c'est-à-dire qu'un attribut est hérité pour un symbole X s'il exprime des propriétés qui dépendent d'autres symboles que ceux apparaissant dans les dérivations de X.

Pour chaque règle de production de la grammaire on définit un ensemble de règles sémantiques qui définissent :

- 1 - la valeur des attributs synthétisés du symbole non terminal de la partie gauche,

- 2 - la valeur des attributs hérités des symboles de la partie droite.

Dans un tel mode de description l'arbre syntaxique d'un programme est "décoré" par la valeur des attributs de chaque symbole, qui expriment l'ensemble des informations sémantiques associées aux noeuds de l'arbre.

Le but de l'évaluation de toutes ces informations est d'obtenir la valeur sémantique du programme : cette valeur est la valeur d'un des attributs synthétisés de l'axiome de la grammaire, attributs associés à la racine de l'arbre syntaxique de ce programme.

1.2.3. b. - Exemple [Lo 75] :

Considérons un langage dans lequel tout programme est constitué d'un ensemble de déclarations, suivi d'une liste d'instructions. Nous donnons partiellement les règles de la grammaire engendrant ce langage : chaque règle étant accompagnée des règles sémantiques déterminant la valeur des attributs associés aux symboles apparaissant dans cette règle.

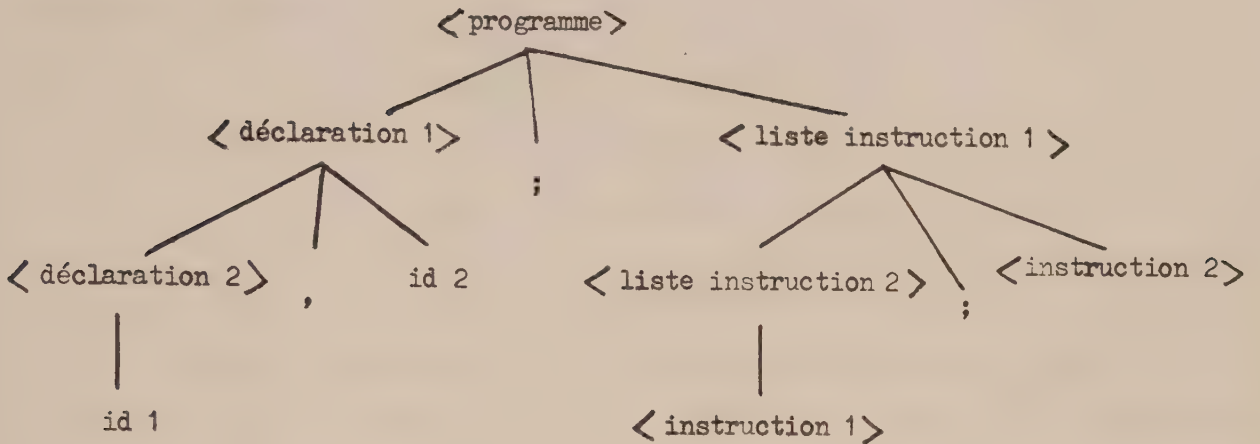
Remarque : pour pouvoir exprimer ces règles sémantiques, on numérote les occurrences des symboles apparaissant dans les règles syntaxiques.

Les attributs utilisés sont :

- init , attribut hérité, délivre le numéro de la première opération dans l'interprétation de l'instruction considérée.
- def , attribut synthétisé, délivre l'ensemble des identificateurs déclarés.
- index , attribut synthétisé, délivre le numéro d'ordre dans l'ensemble des identificateurs déclarés.
- contexte, attribut hérité, délivre le contexte d'élaboration d'une instruction, c'est-à-dire l'ensemble des identificateurs, signes opérateurs, etc... valides au moment de cette élaboration.
- texte , attribut synthétisé, délivre la chaîne de caractères constituant le symbole terminal sur lequel il est appliqué.
- fin , attribut synthétisé, délivre le numéro de la dernière opération dans l'interprétation de l'instruction considérée.

Règles syntaxiques	Règles sémantiques
$\langle \text{programme} \rangle :: \langle \text{déclaration} \rangle ;$ $\quad \quad \quad \langle \text{liste instructions} \rangle$	$\underline{\text{contexte}} (\text{liste instructions}) = \underline{\text{def}} (\text{déclaration})$ $\underline{\text{init}} (\text{liste instructions}) = 1$ $\underline{\text{fin}} (\text{programme}) = \underline{\text{fin}} (\text{liste instructions})$
$\langle \text{déclaration 1} \rangle :: \langle \text{déclaration 2} \rangle,$ $\quad \quad \quad \text{id}$	$\text{si } \underline{\text{texte}} (\text{id}) \in \underline{\text{def}} (\text{déclaration 2})$ $\quad \text{alors : erreur ('double définition')}$ $\quad \text{sinon :}$ $\underline{\text{def}} (\text{déclaration 1}) = \underline{\text{def}} (\text{déclaration 2})$ $\quad \quad \quad \underline{\text{texte}} (\text{id})$ $\underline{\text{index}} (\text{déclaration 1}) = \underline{\text{index}} (\text{déclaration 2})$ $\quad \quad \quad + 1$
$\langle \text{déclaration} \rangle :: \text{id}$	$\underline{\text{def}} (\text{déclaration}) = \underline{\text{texte}} (\text{id})$ $\underline{\text{index}} (\text{déclaration}) = 1$
$\langle \text{liste instruction 1} \rangle :: \langle \text{liste}$ $\quad \quad \quad \text{instruction 2} \rangle ;$ $\quad \quad \quad \langle \text{instruction} \rangle$	$\underline{\text{init}} (\text{liste instruction 2}) = \underline{\text{init}} (\text{liste instruction 1})$ $\underline{\text{init}} (\text{instruction}) = \underline{\text{fin}} (\text{liste instruction 2})$ $\quad \quad \quad + 1$ $\underline{\text{fin}} (\text{liste instruction 1}) = \underline{\text{fin}} (\text{instruction})$ $\underline{\text{contexte}} (\text{liste instruction 2}) = \underline{\text{contexte}}$ $\quad \quad \quad (\text{liste instruction 1})$ $\underline{\text{contexte}} (\text{instruction}) = \underline{\text{contexte}} (\text{liste instruction 1})$
$\langle \text{liste instruction} \rangle :: \langle \text{instruction} \rangle$	$\underline{\text{init}} (\text{instruction}) = \underline{\text{init}} (\text{liste instruction})$ $\underline{\text{contexte}} (\text{instruction}) = \underline{\text{contexte}} (\text{liste instruction})$ $\underline{\text{fin}} (\text{liste instruction}) = \underline{\text{fin}} (\text{instruction})$

Considérons l'arbre syntaxique suivant :

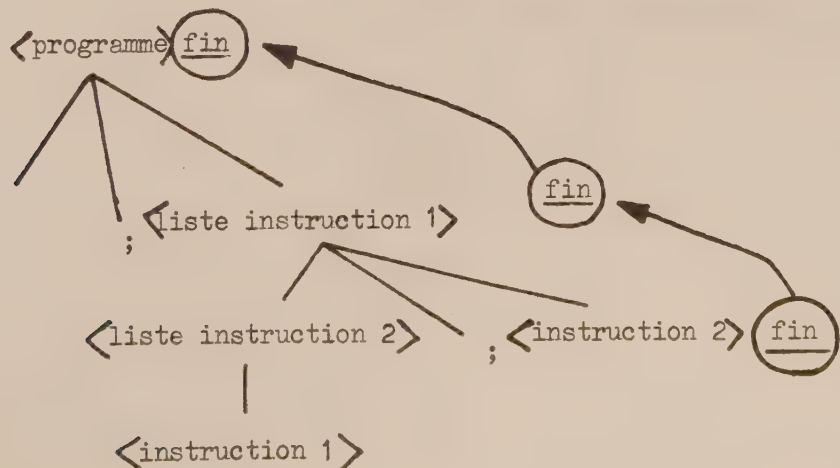


La valeur des attributs sémantiques appliqués à chaque noeud de cet arbre est déterminée par l'application des règles sémantiques:

- par exemple, la valeur de l'attribut synthétisé fin, sur le symbole programme est déterminé par les équations sémantiques :

$$\begin{aligned}
 \underline{\text{fin}} (\text{programme}) &= \underline{\text{fin}} (\text{liste instruction 1}) \\
 &= \underline{\text{fin}} (\text{liste instruction 2}).
 \end{aligned}$$

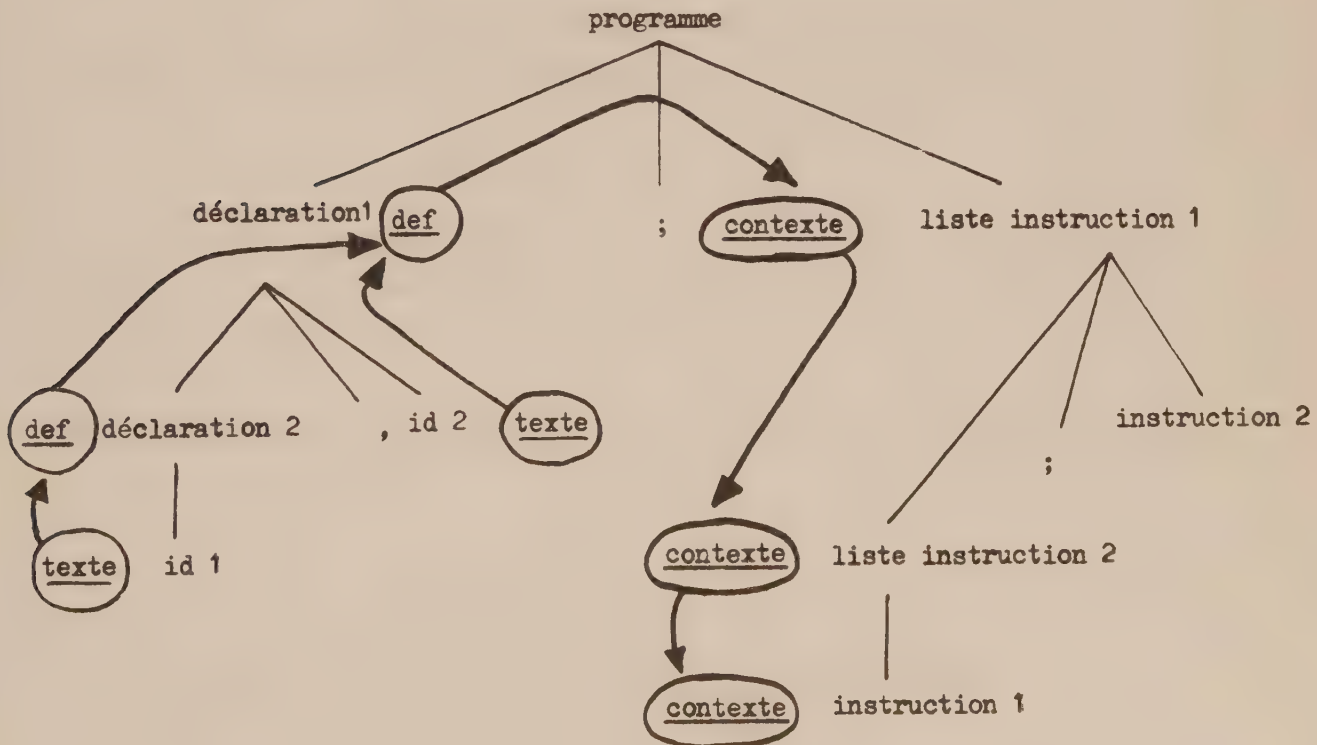
Cette valeur ne dépend que des valeurs des attributs associés aux noeuds qui dérivent du noeud programme :



- la valeur de l'attribut hérité contexte , sur le symbole instruction 1, est déterminé par les équations sémantiques :

$$\begin{aligned}\underline{\text{contexte}}(\text{instruction 1}) &= \underline{\text{contexte}}(\text{liste instruction 2}) \\ &= \underline{\text{contexte}}(\text{liste instruction 1}) \\ &= \underline{\text{def}}(\text{déclaration 1}) \\ &= \underline{\text{def}}(\text{déclaration 2}) \cup \underline{\text{texte}}(\text{id2}) \\ &= \underline{\text{texte}}(\text{id1}) \cup \underline{\text{texte}}(\text{id2})\end{aligned}$$

Cette valeur dépend de la valeur de l'attribut synthétisé def du noeud déclaration 1, valeur qui devient la valeur de l'attribut hérité contexte du noeud liste instruction 1 :



1.2.3. c. - Système d'équations sémantiques :

La construction de l'arbre dynamique associé à un programme Algol 68, telle qu'elle est présentée dans le rapport de définition de ce langage, ou l'élaboration de l'arbre de contrôle de l'interpréteur abstrait du formalisme de Vienne, sont des méthodes d'interprétation sémantique de ce programme. Dans la méthode des attributs, cette interprétation est exprimée par la "décoration" de l'arbre syntaxique, qui exprime la valeur sémantique de sa racine, en fonction des valeurs sémantiques des constituants de l'arbre, en commençant par les feuilles. On exprime cette interprétation par un système d'équations sémantiques :

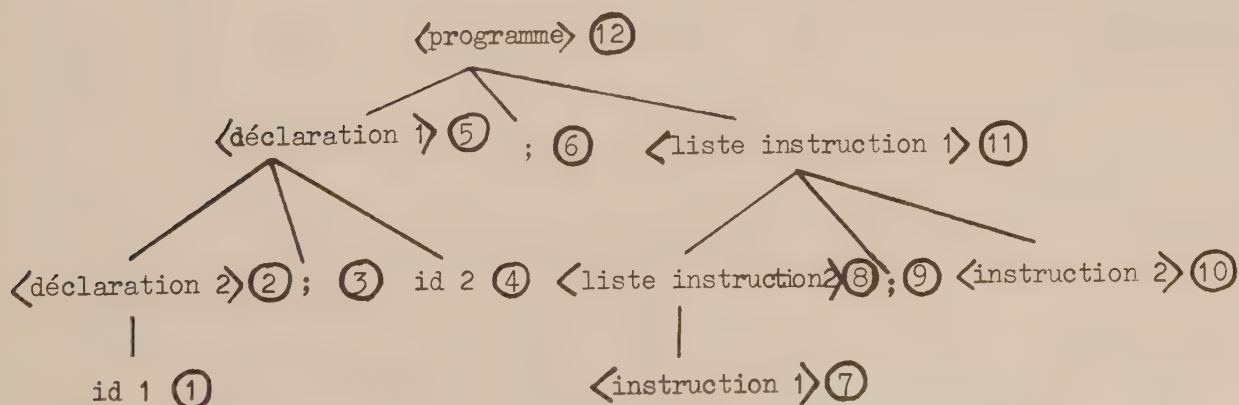
On numérote les noeuds de l'arbre syntaxique où s'appliquent des attributs : on note $\underline{a}(k)$ la valeur de l'attribut sémantique $\underline{a}$ sur le noeud numéro k : on obtient alors un système dit d'équations sémantiques, élaboré de la façon suivante :

- si X est un symbole terminal, constituant la feuille numéro k dans l'arbre syntaxique, le système sémantique est réduit à l'équation :

$$\underline{\text{texte}}(k) = X$$

- si X est un symbole non terminal, le système sémantique associé à l'arbre de racine X est obtenu par la réunion des systèmes associés aux sous-arbres issus du noeud X .

Exemple : on numérote les noeuds de l'arbre précédent de la manière suivante :



La "décoration" de cet arbre, s'exprime, de manière incomplète (on n'écrit pas, en particulier les équations sémantiques des symboles terminaux) par le système d'équations sémantiques suivant :

$$\begin{aligned}\underline{\text{contexte}} \quad (11) &= \underline{\text{def}} \quad (5) \\ \underline{\text{init}} \quad (11) &= 1 \\ \underline{\text{fin}} \quad (12) &= \underline{\text{fin}} \quad (11) \\ \underline{\text{def}} \quad (5) &= \underline{\text{def}} \quad (2) \cup \underline{\text{texte}} \quad (4) \\ \underline{\text{index}} \quad (5) &= \underline{\text{index}} \quad (2) + 1 \\ \underline{\text{def}} \quad (2) &= \underline{\text{texte}} \quad (1) \\ \underline{\text{index}} \quad (1) &= 1 \\ \underline{\text{init}} \quad (8) &= \underline{\text{init}} \quad (11) \\ \underline{\text{init}} \quad (10) &= \underline{\text{fin}} \quad (8) + 1 \\ \underline{\text{fin}} \quad (11) &= \underline{\text{fin}} \quad (10) \\ \underline{\text{contexte}} \quad (8) &= \underline{\text{contexte}} \quad (11) \\ \underline{\text{contexte}} \quad (10) &= \underline{\text{contexte}} \quad (11) \\ \underline{\text{init}} \quad (7) &= \underline{\text{init}} \quad (8) \\ \underline{\text{contexte}} \quad (7) &= \underline{\text{contexte}} \quad (8) \\ \underline{\text{fin}} \quad (8) &= \underline{\text{fin}} \quad (7)\end{aligned}$$

1.2.3. d. - Conclusion :

Dans cette méthode, l'interprétation d'un programme est réalisée en deux phases :

- l'analyse syntaxique, où le système d'équations sémantiques est construit par la définition des équations sémantiques associées à chaque noeud numéroté de l'arbre,

- l'évaluation sémantique, où ce système est résolu.

Cependant, le plus souvent, pour les traducteurs dirigés par la syntaxe, on choisit de ne pas complètement dissocier les deux phases et de réaliser l'évaluation sémantique au fur et à mesure d'une analyse syntaxique, généralement de type descendante déterministe gauche - droite.

La seconde phase pose le problème de la méthode à adopter pour la résolution du système, de façon à la rendre pratiquement efficace : une telle méthode suppose :

- d'une part, une possibilité de réduction du nombre des attributs : on voit sur le seul exemple très simple proposé, que ce nombre est un des principaux handicaps de la méthode des attributs,
- d'autre part, une exploitation optimale des diverses relations de dépendance existant entre les attributs.

CONCLUSION

Nous avons rappelé dans ce chapitre les méthodes de formalisation sémantique actuellement proposées : on peut se demander à ce point de l'exposé si l'une ou l'autre de ces méthodes convient pour une définition de langage extensible, telle que nous la présentions dans l'introduction.

Pour ce qui concerne les méthodes non interprétatives on a remarqué qu'elles ne semblaient pas devoir être retenues pour une définition de langage et que leur aspect très formel, au demeurant fort séduisant, est davantage adapté aux différents problèmes de vérifications de programmes, voire de compilateurs.

Par contre, les méthodes interprétatives, parce qu'elles représentent de façon relativement précise l'architecture d'une implémentation du langage qu'elles décrivent, semblent mieux adaptées à une définition de langage.

Nous verrons dans le chapitre 2 les propriétés d'extensibilité dans les langages telles qu'on peut envisager de les définir à partir de la considération des méthodes interprétatives : nous devons, pour cela, ne pas nous

contenter du modèle simplifié que nous présentions au paragraphe 1.2., et qui consiste en la définition formelle d'une machine abstraite opérant sur les différentes structures du langage.

La simplicité de ce modèle ne rend, en effet, pas compte de la distinction essentielle dans une implémentation réelle, entre ce qui, dans un programme, peut être traité de façon statique et ce qui doit l'être de façon dynamique. Aussi peut-on préciser ce schéma, à la lumière des trois exemples pratiques étudiés, en faisant apparaître deux phases successives, ou, dans la pratique, plus ou moins dissociées, dans le traitement qui précède l'interprétation [Vi 74] ou [Jo 75].

- la première , d'analyse syntaxique, qui à partir d'un programme source exprimé dans la syntaxe du langage, que nous appellerons la syntaxe concrète, élabore un arbre syntaxique,

- la seconde, dite d'analyse sémantique statique [Gr 73], où sont prises en compte les conditions de déclarations et d'utilisation des objets :

- c'est la prise en compte dans la construction des règles de grammaire Algol 68 des conditions de contexte déterminées par les métanotations figurant dans les hyperrègles (les deux phases sont alors mêlées),

- c'est également le principe de construction des objets abstraits du langage de Vienne, à partir d'un programme concret,

- c'est enfin l'écriture du système d'équations sémantiques associé à un arbre syntaxique dans la méthode des attributs.

Le programme considéré a alors donné lieu à la construction d'un ensemble de structures qui constituent ce qu'on appelle la syntaxe abstraite du langage, structures adéquates pour une interprétation, exprimée de façon algorithmique, et qui concerne la sémantique dite dynamique :

- c'est la construction de l'arbre dynamique pour l'interprétation d'un programme Algol 68,

c'est également la définition de l'interpréteur abstrait, et en particulier la construction de son arbre de contrôle, dans le formalisme de Vienne,

c'est enfin la résolution du système d'équations sémantiques dans la méthode des attributs.

Nous allons voir les conséquences d'un tel schéma de description sémantique interprétative sur la définition d'un langage extensible.

CHAPITRE 2

NOTION DE LANGAGE EXTENSIBLE

CHAPITRE 2

NOTION DE LANGAGE EXTENSIBLE

2.1. - DEFINITION DE L'EXTENSIBILITE

2.1.1. - Introduction

Pour résoudre un problème particulier, l'utilisateur d'un ordinateur doit exprimer ce problème en termes d'objets et traitements accessibles par l'intermédiaire du langage de programmation utilisé : si cela est possible, nous dirons que le langage est adapté au problème considéré.

La variété des problèmes soumis à un traitement automatique est telle que se pose, pour tout problème particulier la question de l'existence d'un langage adapté. La solution la plus élémentaire consisterait naturellement à dire : "si le langage n'existe pas, inventons le". Elle conduit rapidement à une prolifération anarchique et coûteuse des langages, dont l'inadaptation n'est plus à démontrer.

On retient plus volontiers deux types de solutions :

- la première consiste en la définition d'un langage universel ("the shell approach" dans [Ch F 68]), prétendant permettre l'accès à tous les types d'objets et traitements requis pour toute catégorie de problèmes : c'est la vocation de langages comme PL/1 ou ALGØL 68.

On conçoit facilement qu'alors l'utilisateur a la désagréable obligation de connaître la définition d'un langage compliqué dont il n'utilise en fait que quelques ressources et dont l'utilisation peut en outre s'avérer fort coûteuse quant à la taille du compilateur nécessaire.

- la seconde solution consiste en la définition d'un langage minimal ("the core approach") ayant la propriété de pouvoir "s'étendre" : ceci signifie qu'un utilisateur a la possibilité d'en modifier automatiquement la définition et d'engendrer, ainsi, le compilateur du langage particulier, adapté à son problème. C'est cette propriété qui est désignée par le terme d'extensibilité, et on appelle langage extensible un langage qui la possède.

2.1.2. - Principes de l'extensibilité

Le principe général de l'extensibilité d'un langage est la possibilité pour un utilisateur de ce langage de définir, à partir des objets et traitements accessibles par le langage, un nouvel ensemble d'objets et traitements, et que cet ensemble soit reconnu par un compilateur construit automatiquement.

La définition d'un langage extensible tel qu'il est généralement envisagé, est constituée de deux éléments [Jo B 72], [Jo 75] ou [Sc J 70].

- un langage de base : considéré comme un assemblage des primitives indispensables, organisées de telle manière qu'elles constituent un langage cohérent.

- un ensemble de mécanismes d'extension, opérant sur des éléments du langage et en engendrant de nouveaux.

On appelle langage étendu, un langage défini par un ensemble d'applications de mécanismes d'extension, à partir du langage de base.

La définition du processus d'extensibilité est généralement récursive, en ce sens que tout langage étendu peut servir à son tour de langage de base pour la génération d'un nouveau langage étendu.

2.1.3. - Types d'extensibilité introduits par les méthodes interprétatives

L'extensibilité telle qu'elle est introduite jusqu'alors dans les langages est définie à partir des conceptions interprétatives des définitions de ces langages, conceptions dégagées au chapitre 1 : celles-ci ont conduit à considérer trois mécanismes disjoints d'extensibilité concernant respectivement les trois phases du traitement interprétatif distinguées dans la conclusion du chapitre 1 : ces mécanismes sont

- l'extensibilité de la syntaxe
- l'extensibilité de la sémantique statique
- l'extensibilité de la sémantique dynamique.

2.1.3. a. - L'extensibilité de la syntaxe :

On dit qu'une extension est syntaxique si le champ d'application de ses mécanismes est la syntaxe concrète du langage : une telle extension a pour but de modifier l'aspect externe du langage par l'introduction de nouvelles phrases dont la forme est adaptée au domaine d'applications auquel est destiné le langage.

Un système d'extensions syntaxiques est généralement réalisé par un ensemble de déclarations exprimant la forme de la nouvelle phrase utilisable, le contexte possible de son utilisation, et la signification de celle-ci, en termes de phrases préalablement définies.

Nous verrons au paragraphe 2.2. la réalisation de ces extensions sous la forme de macros syntaxiques.

2.1.3. b. - L'extensibilité de la sémantique statique :

L'extension de la sémantique statique consiste en la définition de nouvelles catégories d'objets susceptibles d'être pris en compte par l'interpréteur. Ce mécanisme d'extension est réalisé par un système de déclarations spécifiant la configuration et les règles d'utilisation de ces objets. Nous verrons au paragraphe 2.3. des exemples de réalisation d'extension de la sémantique statique dans quelques langages actuels.

2.1.3. c. - L'extensibilité de la sémantique dynamique :

Dans une conception interprétative, l'extensibilité de la sémantique dynamique a pour domaine d'application la syntaxe abstraite du langage, et consiste en la possibilité de modification de son interpréteur. Ainsi dans une description du type de Vienne une extension de la sémantique dynamique consiste à définir de nouvelles instructions du répertoire de l'interpréteur abstrait, en construisant la routine d'interprétation associée.

A titre d'exemple, étant donné un certain langage, supposons que soit créée, par le mécanisme d'extension de la sémantique statique, une nouvelle catégorie d'objets : la déclaration d'un objet appartenant à cette catégorie active une routine du répertoire de l'interpréteur, routine qui a dû être ajoutée à ce répertoire, et qui doit être assimilée aux autres routines de l'interpréteur, c'est-à-dire reconnue et exécutée directement.

Autre exemple, supposons qu'un certain type de tableaux soit créé par une extension de la sémantique statique : il faut alors étendre le répertoire de l'interpréteur en y incluant, par exemple, une routine permettant l'interprétation de la sélection d'un des éléments d'un tableau de ce type.

Pour une telle extension il est donc nécessaire d'intervenir au niveau même de l'implémentation du langage, et il semble qu'actuellement aucun formalisme ne rende un tel processus automatique.

Nous reviendrons sur ce problème dans la conclusion de ce chapitre.

2.2. - EXTENSIBILITE SYNTAXIQUE PAR MACROS

2.2.1. Principe

La déclaration de macro syntaxique est le procédé le plus généralement présenté d'extension syntaxique. Le principe de ce procédé est la génération de nouvelles productions terminales de la grammaire : ce procédé sous entend donc que le langage contient une fonction permettant d'introduire dynamiquement des transformations de la grammaire engendrant le langage.

De la façon la plus générale, cette fonction s'exprime sous la forme d'une déclaration de macro syntaxique du type suivant [Sc J 70] ou [Vi 74].

macro ϕ means ρ

où ϕ est une règle de production qui spécifie la présentation syntaxique des nouvelles productions terminales.

et ρ est l'interprétation sémantique associée à ϕ , qui en spécifie la signification.

L'effet d'une macro syntaxique est d'incorporer la règle définie, à la grammaire du langage, accompagnée de la sémantique spécifiée dans la partie signification de la macro.

2.2.2. Exemple et conséquences

Etant donné un langage où chaque instruction est une production terminale du symbole $\langle \text{STATEMENT} \rangle$, on définit une macro syntaxique introduisant un nouveau type d'instruction, l'incrémentation, par la déclaration

```
macro  $\langle \text{STATEMENT} \rangle ::$  inc  $\langle \text{IDENTIFIER} \rangle$  means  $\langle \text{IDENTIFIER} \rangle := \langle \text{IDENTIFIER} \rangle$   
+ 1
```

On peut considérer que, du point de vue utilisation, ces déclarations de macro syntaxique sont identiques à des déclarations de procédure dont la partie signification constitue le corps. De fait, le symbole non terminal $\langle \text{IDENTIFIER} \rangle$ peut être considéré comme un paramètre formel, et l'apparition dans le texte d'un programme, soumis à une analyse syntaxique, d'une construction du type : inc X, constitue un appel de cette macro syntaxique : la sémantique de cet appel consiste en la compilation d'une copie de la partie signification de la macro, où X est substitué à $\langle \text{IDENTIFIER} \rangle$.

Il faut cependant remarquer qu'à la différence d'une procédure, l'évaluation d'un appel de macro est faite à la compilation : c'est ainsi que, suivant ce procédé, la compilation d'un programme écrit dans un langage utilisant une extension syntaxique, se fait en deux étapes :

- une évaluation des appels des macros syntaxiques nécessaires
- un assemblage du programme résultant.

L'utilisation de macros syntaxiques comme procédé d'extensibilité ne se conçoit que si une telle définition est décrite par la grammaire du langage de base. Ceci impose trois conditions essentielles :

- la définition d'une macro syntaxique doit s'exprimer comme une déclaration.

- la partie signification d'une macro syntaxique doit s'exprimer comme une séquence d'instructions écrites dans la syntaxe du langage de base.

- les paramètres formels apparaissent dans ces déclarations étant des symboles non terminaux de la grammaire du langage de base, celle-ci doit inclure un mécanisme autorisant tout symbole non terminal, à devenir, sous certaines conditions, élément d'une production terminale.

2.2.3. Restrictions au procédé

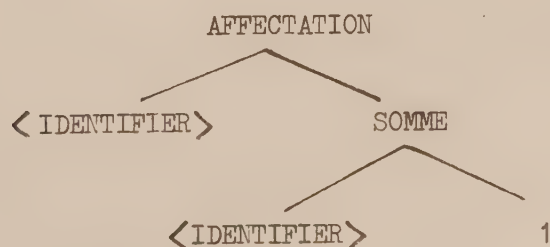
2.2.3. a. - Il semble que dans ce procédé, il ne soit pas possible de définir, dans une seule macro, plusieurs règles de productions, ce qui interdit, en particulier, toute définition d'une règle récursive.

2.2.3. b. - D'autre part, l'aspect général du procédé, en ce sens que tout symbole non terminal peut y être utilisé, nuit grandement à son efficacité :

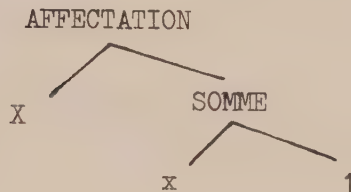
Une première solution fut apportée par LEAVERWORTH [Le 66] en limitant l'ensemble des symboles non terminaux autorisés dans une définition de macro syntaxique, ainsi que les traitements sur ces symboles.

D'autres solutions proposent d'extraire l'information fournie par la définition de l'extension, par une analyse de celle-ci, et si possible de vérifier sa correction. Nous étudions deux de ces solutions :

- Lors de la compilation de la définition d'une macro syntaxique, on peut [Ha71] établir un schéma, constitué de l'ensemble des routines qui seront activées par l'appel de cette macro. Ainsi dans l'exemple du paragraphe 2.2.2., l'analyse de la macro syntaxique produira un schéma de compilation de la forme :



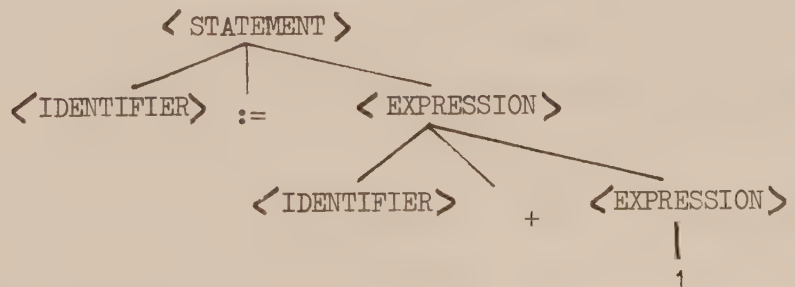
Son utilisation pour la compilation d'une construction inc X consiste en l'exécution des routines SOMME et AFFECTATION avec les arguments appropriés.



- Plus récemment VIDART [Vi 74] propose une véritable analyse syntaxique de la partie signification d'une macro, avec élaboration d'un programme abstrait : pour cela on impose que cette partie soit une dérivation du non-terminal figurant dans la partie gauche de la règle de production de la macro.

Cette méthode accroît la fiabilité et l'efficacité de l'évaluation des appels : la fiabilité, parce que des détections d'erreurs ont pu être réalisées à la définition même ; l'efficacité, parce que, comme précédemment, il n'y a pas répétition de l'analyse, même pour plusieurs utilisations d'une même extension.

Ainsi pour l'exemple du paragraphe 2.2., l'arbre syntaxique construit pourrait être :



Le programme abstrait associé, a, comme dans le schéma proposé dans [Ha 71] deux feuilles indéterminées, correspondant à l'évaluation d'une dérivation de <IDENTIFIER> .

Dans l'utilisation de cette extension, la compilation d'une instruction d'incrémentatation :

inc X

consiste en un appel de la macro et produit le programme abstrait associé à l'évaluation de X, pour préciser ces feuilles indéterminées.

2.3. - EXEMPLES D'EXTENSIBILITE DE LA SEMANTIQUE STATIQUE

Nous donnons l'exemple de trois langages, ECL [We 71], SIMULA 67 [DaM66] et ALGØL 68 [Gr A 75], qui possèdent un mécanisme d'extension de la sémantique statique autorisant la définition de nouvelles catégories de valeurs par des déclarations faisant intervenir les objets suivants :

- un ensemble de catégories de valeurs primitives, appelées modes (dans ECL et ALGØL 68) ou classes (dans SIMULA 67) qui fait partie de la définition du langage.
- un ensemble de constructeurs qui agissent sur ces modes (ou classes) pour délivrer un autre mode (ou classe), assurant ainsi la récursivité du mécanisme.

2.3.1. Les types de valeurs dans le système ECL

Le système ECL, développé à l'Université de Harvard consiste en un langage appelé EL 1 et un système construit autour de ce langage pour en assurer l'utilisation.

EL 1 a, selon ses concepteurs, la propriété d'être un langage extensible : il a en particulier des propriétés d'extension dans les domaines suivants : syntaxe, types de valeurs, opérations.

- l'extension syntaxique permet la spécification et l'interprétation de nouvelles représentations.

- l'extension sur les types de valeurs permet au programmeur de définir de nouveaux types de valeurs, par un mécanisme de déclaration.

- l'extension des opérations permet de les faire agir sur les valeurs dont le type est obtenu par extension aussi bien que sur les valeurs primitives.

La définition de nouveaux types de valeurs, ou modes, repose sur la propriété suivante : les modes sont des valeurs du langage ; leur mode est MODE.

Tout nouveau type de valeurs, c'est-à-dire toute nouvelle valeur de mode MODE est obtenue à partir de routines de construction de modes comme ROW, STRUCT ou PROC, etc...

Un mécanisme de déclaration en permet la reconnaissance à travers une nouvelle chaîne de caractères :

```
Exemple : [Br 71]      triple ← ROW (3, INT) ;  
                      record ← STRUCT (id : STRING, history : triple,  
                                       flag : BOOL)
```

INT, STRING et BOOL sont des modes primitifs, le deuxième champ de la structure est une valeur sélectionnée par history et de mode triple.

2.3.2. Les classes dans SIMULA 67

Le mécanisme d'extension comprend en particulier, dans SIMULA 67, la possibilité d'introduction de nouveaux types de valeurs et d'opérations sur ces valeurs : il est réalisé par la déclaration de classe, qui définit une classe d'objets, comme un ensemble de valeurs possédant un même ensemble d'attributs fonctionnels [Jc 71] . Ces attributs sont exprimés dans le corps de la déclaration de classe :

```
Exemple [An71]: class POINT (X, Y) ; real X, Y ;  
                begin real RO, TETA ;  
                    RO := SQRT (X ↑ 2 + Y ↑ 2) ;  
                    TETA := ARCTAN (X / Y)  
                end ;
```

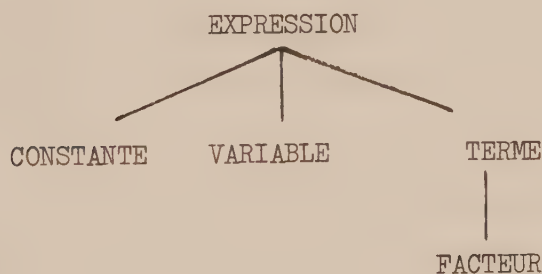

Cette classe, équivalente à la définition d'une structure à deux champs réels en ALGØL 68, a deux attributs : RO et TETA, définis dans le corps de la déclaration. Ils déterminent les coordonnées polaires du point de coordonnées cartésiennes X et Y, et peuvent être utilisés comme sélecteurs d'une variable P de type ref (POINT).

En plus de la propriété de récursivité, propre à tout mécanisme d'extensibilité, qui permet d'utiliser toute classe comme un type de valeurs primitif, SIMULA 67 a une propriété de modularité dans la définition de sous classes de valeurs :

Par exemple, la classe EXPRESSION étant définie on peut déclarer les sous classes suivantes :

```
EXPRESSION class  CONSTANTE (K) ; real K ; ... ;  
EXPRESSION class  VARIABLE (ID) ; text ID ; ... ;  
EXPRESSION class  TERME   (X, Y) ; ref (EXPRESSION) X, Y ; ... ;  
TERME class  FACTEUR ; ... ;
```

qui se placent sur un arbre qui exprime que pour chaque sous-classe, sont définis les attributs fonctionnels de toute classe qui la précède dans l'arbre :



2.3.3. Les déclarations de mode ALGØL 68

Toute valeur ALGØL 68 est d'un certain mode. Un mode est la généralisation du type de valeurs que l'on trouve, par exemple, dans ALGØL 60 : il rassemble des valeurs sur lesquelles les mêmes opérations et relations sont définies.

Certains modes appartiennent à la définition du langage de base :

- ce sont les modes primitifs : ent , réel, bool, car, format

- ou les modes standards, construits à partir d'eux par les constructeurs rep, struct, [] , proc ou union : chaîne, compl sont des exemples de modes standards (le premier est un rang de caractères et le second une structure à deux champs réels sélectionnés respectivement par "re" et "im").

D'autres modes peuvent être définis par le programmeur, à l'aide de ces mêmes constructeurs agissant sur des modes primitifs, standards ou préalablement définis, dans une déclaration de mode :

Ainsi peut-on définir en ALGØL 68 un nombre infini de modes, c'est-à-dire un nombre infini de classes de valeurs : ce mécanisme est, comme dans les langages précédemment cités, un mécanisme d'extensibilité de la sémantique statique.

La déclaration de mode permet de représenter par un nouveau symbole, appelé indicateur de mode, un mode élaboré à partir des constructeurs :

Exemple : après la déclaration,

mode livre : struct (chaîne titre, rep livre suivant)

la chaîne de caractères livre représente un mode ALGØL 68 qui est une structure à deux champs : le premier, sélectionné par "titre" est de mode chaîne, et le second, sélectionné par "suivant" est de mode rep livre.

On remarque sur cet exemple que les indicateurs de mode ainsi déclarés jouissent, comme dans les langages précédemment cités, des mêmes prérogatives que les autres indicateurs : ils peuvent apparaître en partie droite d'une déclaration de mode (y compris, sous certaines conditions dans leur propre déclaration), en assurant ainsi la récursivité du mécanisme d'extensibilité.

Toutefois le domaine de validité de leur déclaration est limitée à la région de programme où elle se situe, et une occurrence d'utilisation d'un tel indicateur de mode ne pourra apparaître que dans cette région.

Remarquons d'autre part, que ce mécanisme de déclarations de mode est moins puissant que celui des déclarations de classe SIMULA 67 puisque il ne permet pas une définition équivalente à celle des attributs fonctionnels.

CONCLUSION

La définition de l'extensibilité, suivant une conception interprétative de la description d'un langage conduit, on vient de le voir, à l'introduction de trois mécanismes distincts d'extensibilité :

- un mécanisme réalisant des extensions syntaxiques,
- un mécanisme réalisant des extensions de la sémantique statique,
- enfin un mécanisme réalisant des extensions de la sémantique dynamique.

Si l'on peut admettre que les deux premiers ont mené à des applications dans quelques langages (le procédé de déclaration de nouveaux modes ou d'opérateurs est aujourd'hui familier des programmeurs avertis), il faut considérer que le troisième n'est pas encore suffisamment appréhendé : il n'est cependant pas douteux que des résultats dans ce domaine conduiront à une définition correcte des langages extensibles, qui rendra leur utilisation efficace.

Quel est donc le problème de l'extension de la sémantique dynamique, dans cette conception interprétative ?

Il s'agit de la modification de l'interpréteur du langage, c'est-à-dire d'une construction de nouvelles routines d'interprétation, qui les assimile aux routines de base.

On se rend compte alors que ce processus doit être intimement lié au modèle d'implémentation du langage de base, c'est-à-dire que le problème de la modification de l'interpréteur est identique à celui de sa définition.

Si l'on veut qu'un tel processus soit, dans un langage extensible, automatique il faut donc envisager que la construction de l'interpréteur le soit.

Remarquons enfin qu'il n'existe pas, à notre connaissance, de procédé d'extensibilité de la sémantique statique assez puissant pour permettre d'engendrer, par exemple, à partir d'un langage où est définie une classe d'objets sans possibilité de contrôle statique du mode (du type "pointeurs PL/1") un autre langage où serait définie une classe d'objets avec possibilité de contrôle statique du mode (du type "repères ALGØL 68").

Nous proposons un modèle de formalisation de la sémantique, suffisamment proche d'un schéma d'implémentation, pour qu'elle permette, à notre avis, non seulement la définition d'un ensemble extensible de classes d'objets, (y compris des extensions de ce dernier type) mais aussi la définition formelle d'une construction automatique d'algorithmes de traitement.

C'est ce modèle que nous présentons dans le chapitre suivant.

CHAPITRE 3

MODELE DE FORMALISATION

CHAPITRE 3

MODELE DE FORMALISATION

"... Nous dirons donc que, pour le coupe-papier, l'essence -c'est-à-dire l'ensemble des recettes et des qualités qui permettent de le produire et de le définir- précède l'existence..."

J.P. SARTRE

INTRODUCTION -

Un langage étant défini comme un ensemble de phrases, un programme écrit dans un langage de programmation constitue une phrase particulière de ce langage : cette phrase est l'expression syntaxique d'un certain objet qui peut être pris en compte, d'une façon décrite par la sémantique du langage, par une certaine machine : on appelle exécution du programme par cette machine, la prise en compte de cet objet.

On dit que la phrase considérée donne accès à cet objet.

L'exécution du programme fait intervenir d'autres objets, également **accessibles** par des phrases du langage (par exemple, ce qu'on appelle les "valeurs" dans certaines définitions de langage).

L'ensemble des objets auquel les phrases d'un langage donnent accès est caractéristique de ce langage.

Le modèle de formalisation que nous proposons dans ce chapitre, et qui nous semble convenir à la définition extensible d'un langage, repose sur une définition des propriétés caractéristiques de chaque catégorie d'objets auxquels donnent accès les phrases de ce langage.

Plus précisément nous dirons qu'un objet possède un certain ensemble de propriétés potentielles qui définissent ce que cet objet peut subir ou produire : dans l'ensemble des objets, ceux qui ont le même ensemble de propriétés sont dits constituer une classe d'objets.

La définition formelle d'une classe d'objets comprend l'expression d'un certain nombre d'actions, dites primitives, qui définissent les propriétés de cette classe : cette expression est engendrée par une grammaire, dite grammaire conceptuelle, qui décrit également l'aspect syntaxique des objets appartenant à la classe définie.

La définition d'un objet est alors la déclaration de son appartenance à une certaine classe, et consiste en l'exécution de certaines des actions primitives faisant partie de la définition de cette classe, exécution selon un certain algorithme qui définit le premier traitement sur l'objet : sa création. Cet algorithme est décrit, comme tous les algorithmes de traitements, par une grammaire dite grammaire opérationnelle.

Nous nous limitons dans cet exposé à l'étude des objets de type "valeur".

3.1 - LES ACTIONS PRIMITIVES -

Nous définissons les propriétés potentielles caractérisant chaque classe d'objets à partir de quatre actions primitives, qui sont : l'allocation, l'écriture, la lecture et la désignation.

Lorsqu'elles sont exécutées, ces actions font intervenir une mémoire, définie comme un ensemble de "zones en mémoire".

On distingue deux types de zones en mémoire :

- les zones, que nous appellerons zones dominées, sur chacune desquelles il est possible d'exécuter une écriture un nombre arbitraire de fois,
- les zones, que nous appellerons zones dominantes, sur chacune desquelles il n'est possible d'exécuter qu'une et une seule écriture.

On pourrait considérer ces zones en mémoire sur un ordinateur, comme un certain ensemble de mots mémoires qui soient adressables par le

programmeur, dans le premier cas, et non adressables, dans le second. Cette représentation physique n'est naturellement pas à considérer à la lettre pour une implantation mémoire, optimisée, du langage, mais elle est commode pour l'exposé : remarquons cependant que si ces zones ne sont pas nécessairement installées physiquement en mémoire, leur rôle sémantique est réel.

Ainsi, dans l'implémentation d'une variable FØRTRAN, on peut considérer un élément de la table d'identificateurs comme une zone dominante, la zone dominée correspondante contenant alors la valeur de la variable.

De même si on se réfère au modèle d'implémentation du langage ALGØL 68 présenté dans [Gr A 75] on peut se représenter physiquement les zones dominées comme des éléments de la pile des générateurs locaux ou du tas, tandis que les zones dominantes sont en particulier les éléments de la pile d'identificateurs. Nous remarquerons cependant, à propos de quelques exemples (Cf. § 4 . 2) que de telles implémentations ne retiennent parfois que le rôle sémantique d'une zone et non sa présence en mémoire.

3.1.1 - L'allocation -

L'allocation est une action primitive qui consiste à attribuer une zone en mémoire à une autre. Il y a deux types d'allocations :

- le premier qui attribue une zone dominée à une zone dominante,
- le second qui attribue une zone dominante à une autre zone dominante.

Pour rappeler que dans certaines implémentations ces allocations peuvent avoir lieu, respectivement, à l'exécution et à la compilation, nous appellerons allocation dynamique la première, et allocation statique la seconde.

3.1.2 - L'écriture -

L'écriture sur une zone en mémoire consiste en la mise en place dans cette zone d'un objet, chaque type d'objet définissant un type d'écriture. Nous distinguerons les types d'écriture suivants :

3.1.2.a - L'écriture-v -

L'écriture-v est la mise en place d'une valeur classique. Ainsi dans l'instruction FØRTRAN :

I = 3

si la notation 3 apparaît pour la première fois dans le programme, la déclaration implicite de la constante 3 provoque l'écriture de la valeur entière 3 sur une zone dominante, élément de la table des constantes.

3.1.2.b - L'écriture-a -

L'écriture-a est la mise en place d'une adresse : cette écriture, lorsqu'elle est exécutée sur une zone dominante, peut être associée à une allocation dynamique d'une zone dominée à cette zone dominante.

On peut donner pour exemple de telle action l'implémentation des identificateurs ALGØL 60, telle qu'elle est présentée dans [Bo 64] : une première phase du programme compilateur (passage EDITEUR) construit un élément de chaîne codée, associée à chaque identificateur, et mémorise dans une mémoire structurée en pile, le "profil qualitatif" de cet identificateur comportant un numéro permettant la génération, au cours de la seconde phase de compilation, des réservations de mémoire et des champs variables des instructions relatives à cet identificateur.

3.1.2.c - L'écriture-d -

L'écriture-d est la mise en place des paramètres d'une fonction d'accès (ou descripteur) à une valeur composée, c'est-à-dire une valeur de type tableau ou structure de certains langages de type ALGØL.

Lorsqu'elle est exécutée sur une zone dominante, cette écriture est associée à une allocation statique, à celle-ci, d'une autre zone dominante.

Ainsi dans la déclaration ALGØL 68 :

[] ent t = (2,3,4)

l'écriture-d installe le descripteur en mémoire et établit l'allocation statique de la zone des valeurs.

3.1.2.d - L'écriture-f -

L'écriture-f est la mise en place des paramètres d'un format d'échange de valeur : c'est-à-dire la spécification et la présentation de cette valeur.

Il s'agit par exemple, en FØRTRAN, des paramètres F5.2 spécifiant le type de valeur échangée et sa présentation. Nous reviendrons plus en détail sur l'utilisation de cette action primitive au paragraphe 3.4.

3.1.2.e - L'écriture-n -

L'écriture-n est la mise en place d'un objet qui est associé à une notion de la grammaire conceptuelle. L'utilisation des classes d'objets possédant parmi leurs propriétés potentielles, celle d'écriture-n, sera vue au chapitre 5.

3.1.3 - La lecture -

A chaque type d'écriture correspond un type de lecture : la lecture d'une zone en mémoire consiste à considérer son contenu pour une utilisation. Cette action peut être exécutée un nombre arbitraire de fois sur tout type de zone.

3.1.4 - La désignation -

L'action primitive de désignation consiste en la mise en place d'une fonction d'accès à une zone dominante : cette action peut être exécutée qu'une seule fois sur une zone dominante et réalise une fonction de l'ensemble des objets externes (au sens de [Gr A 72]) dans l'ensemble des objets :

- dans le cas où l'objet externe est un identificateur, on appelle identifier, l'action de désigner,
- dans le cas où l'objet externe est une notation, on appelle dénoter, l'action de désigner,

- dans le cas où l'objet externe est une formule, on appelle formuler, l'action de désigner.

Remarque 3.1.4.a -

De même qu'il existe des notations de valeurs, comme 3 ou 3.14, il existe des notations d'adresse, qui correspondent, dans une implantation mémoire du type de celle présentée dans [Gr A 75] à des emplacements mémoires du tas. Il existe également des notations de descripteur, que l'on trouve, par exemple, sous la forme des propositions collatérales ALGØL 68. Il existe des notations de format, comme les modèles de formats en ALGØL 68. Il existe enfin des notations de notions, qui sont les productions terminales de celles-ci dans la grammaire du langage.

Remarque 3.1.4.b -

Comme les valeurs, les descripteurs peuvent être formulés (tranches et sélections) ainsi que les formats, pour lesquels on peut imaginer des opérations de troncature ou de concaténation. Il existe de même pour les adresses, des formules monadiques et diadiques. Dans une implantation mémoire telle que celle présentée dans [Gr A 75], ces formules, dont les opérands correspondent à des emplacements mémoires du tas, **délivrent** l'adresse d'un emplacement du tas, selon des opérations laissées au choix de l'implémenteur, pour respecter son type de gestion mémoire. On peut imaginer, par exemple, la formule monadique :

suiv j qui délivrerait l'adresse de l'emplacement suivant l'emplacement identifié par j.

On pourrait ainsi effectuer un traitement de liste de manière simplifiée.

On peut encore imaginer une formule diadique du type :

j ram 1 qui, dans le tas, délivrerait l'adresse de l'emplacement suivant celui dénoté par 1, et y transférerait la valeur contenue dans l'emplacement identifié par j.

On pourrait utiliser une telle formule dans une procédure de ramasse-miettes.

3.2 - LES DIFFERENTES CLASSES D'OBJETS -

Les actions primitives que nous venons de définir permettent de caractériser les différentes classes d'objets.

Par définition, une classe d'objet comporte, parmi les propriétés potentielles qui la définissent, la donnée d'une zone dominante : pour rappeler que sur une telle zone une action d'écriture n'est exécutable qu'une fois, et une seule, chaque classe d'objets sera appelée constante de genre, où genre est spécifié par le type d'écriture exécutable sur cette zone dominante.

Selon le type d'écriture, on définit les constantes suivantes :

3.2.1 - Constante de valeur -

L'ensemble des propriétés sémantiques caractérisant une constante de valeur est constitué :

- d'une écriture-v exécutable une fois et une seule sur une zone dominante,
- d'une lecture-v exécutable un nombre arbitraire de fois sur cette zone,
- d'une désignation de cette zone.

D'après ce que nous avons vu au § 3.1.2.a : dans l'instruction FØRTRAN :

$$I = 3$$

l'objet désigné par la notation 3 est un objet de classe constante de valeur.

3.2.2 - Constante d'adresse -

L'ensemble des propriétés sémantiques caractérisant une constante d'adresse est constitué :

- d'une écriture-a exécutable une fois et une seule sur une zone dominante,
- d'une lecture-a exécutable un nombre arbitraire de fois sur cette zone,

- d'une désignation de cette zone,
- d'une allocation dynamique d'une zone dominée à cette zone.

Une constante d'adresse permettra par cette dernière allocation de réaliser une variable (Cf. § 4.2.3).

3.2.3 - Constante de descripteur -

L'ensemble des propriétés sémantiques caractérisant une constante de descripteur est constitué :

- d'une écriture-d exécutable une fois et une seule sur une zone dominante,
- d'une lecture-d exécutable un nombre arbitraire de fois sur cette zone,
- d'une désignation de cette zone,
- d'une allocation statique d'une autre zone dominante à cette zone.

Ainsi dans l'exemple du § 3.1.2.c, la constante de descripteur est l'ensemble des propriétés permettant l'accès au tableau désigné par t.

3.2.4 - Constante de format - (Cf. § 3.4)

L'ensemble des propriétés sémantiques caractérisant une constante de format est constitué :

- d'une écriture-f exécutable une fois et une seule sur une zone dominante,
- d'une lecture-f exécutable un nombre arbitraire de fois sur cette zone,
- d'une désignation de cette zone.

3.2.5 - Constante de notion - (Cf. Chapitre 5)

L'ensemble des propriétés sémantiques caractérisant une constante de notion est constitué :

- d'une écriture-n exécutable une fois et une seule sur une zone dominante,

- d'une lecture-n, exécutable un nombre arbitraire de fois sur cette zone,
- d'une désignation de cette zone.

3.3 - LES GRAMMAIRES CONCEPTUELLE ET OPERATIONNELLE -

L'expression des propriétés potentielles caractérisant chaque classe d'objet, telles qu'elles viennent d'être présentées, est engendrée par une grammaire, dite conceptuelle, qui engendre également l'aspect syntaxique des objets appartenant à chacune de ces classes.

Les traitements sur ces objets, à commencer par leur création, sont décrits par un ensemble d'algorithmes engendrés par une grammaire, dite opérationnelle : nous donnons la forme de la grammaire conceptuelle, dont nous verrons un exemple, à propos du langage de base, dans le chapitre 4. Nous nous contenterons d'esquisser la grammaire opérationnelle, nous attachant principalement à en montrer les relations avec la grammaire conceptuelle. Nous présentons enfin une idée sur l'intervention respective de ces deux grammaires, tant du point de vue du programmeur que de celui du compilateur.

3.3.1 - La grammaire conceptuelle -

La grammaire conceptuelle est une grammaire de type context-free [Gi 66], dans laquelle on distingue deux classes de symboles terminaux. Nous la définissons par le quintuplet suivant :

$$G = (N, T, \Sigma, :, \sigma)$$

où N est un ensemble fini de symboles non terminaux, ou notions,

T est un ensemble fini de symboles terminaux syntaxiques,

Σ est un ensemble fini de symboles terminaux sémantiques, chaque symbole exprimant une des actions primitives définies au paragraphe 3.1,

$:$ est une relation de production de N dans $(N \cup T \cup \Sigma)^*$, constituée d'un ensemble fini de règles de grammaire context-free,
et σ est l'axiome de la grammaire, élément privilégié de N.

Remarque 3.3.1.a -

Les ensembles T et Σ sont des ensembles disjoints.

Remarque 3.3.1.b -

Les productions terminales de la grammaire conceptuelle ont une partie syntaxique constituée d'éléments de T , et une partie sémantique constituée d'éléments de Σ , ces deux parties pouvant être imbriquées dans l'ordre lexicographique des productions.

3.3.2 - La grammaire opérationnelle -

Alors que la grammaire précédente ne décrit que des concepts, celle-ci engendre, à partir d'un alphabet fini de relateurs, des algorithmes d'exécution d'actions primitives, appelés opérateurs : étant donné un objet, les opérateurs agissent sur la partie sémantique de la production terminale de la grammaire conceptuelle qui décrit la classe à laquelle appartient l'objet considéré, pour délivrer un ensemble d'informations caractérisant l'état de cet objet. L'évolution de cet état constitue l'élaboration d'un programme.

C'est donc la grammaire opérationnelle qui décrit effectivement la création et le traitement des valeurs, et l'exécution et l'enchaînement des instructions.

Pour illustrer la relation entre ces deux grammaires, nous proposons l'exemple suivant : nous avons vu que l'un des algorithmes engendrés par la grammaire opérationnelle est un algorithme de création, spécifiant, pour l'objet concerné, son appartenance à une certaine classe, exprimée par une production terminale de la grammaire conceptuelle. Cet algorithme qui reprend le mécanisme classique de déclaration, est appliqué aux éléments de l'alphabet sémantique apparaissant dans cette production, et détermine le schéma sémantique de l'implantation de l'objet en mémoire. Très grossièrement, cet algorithme est le suivant :

- exécution de l'action désignation sur une zone dominante,

- exécution d'une action écriture sur cette même zone,
- exécution éventuelle d'une action allocation.

La déclaration est explicite pour les objets identifiés et implicite pour les objets autrement désignés : la déclaration est alors réalisée par une simple reconnaissance syntaxique de la notation ou de la formule.

3.3.3 - L'intervention des grammaires -

Nous présentons schématiquement la mise en oeuvre de cette relation entre les grammaires conceptuelle et opérationnelle.

Du point de vue du programmeur, bienqu'il ne fait apparaître dans le texte du programme, que les parties syntaxiques classiques des objets manipulés, il considère simultanément l'intervention des grammaires conceptuelle et opérationnelle, pour déterminer et contrôler la définition et l'utilisation de chacun de ces objets, en fonction du contexte du programme et du concept auquel il fait appel : c'est-à-dire que, contrairement aux modes classiques de définition des langages, le programmeur n'a plus seulement une idée intuitive, parce que informelle, des fonctions sémantiques associées aux dérivations de la grammaire, mais il peut déterminer formellement la sémantique des phrases du langage qui constituent un programme.

De même, de manière classique, le processus de compilation est dirigé par la seule syntaxe du langage : l'algorithme d'analyse syntaxique à retenir doit être déterministe. Pour ce qui concerne l'intervention de la grammaire conceptuelle dans cette phase on peut utiliser les algorithmes d'analyse descendante, qui supposent une transformation automatique préalable des règles de cette grammaire, en une forme adéquate pour l'algorithme retenu. Le résultat en est l'exhibition d'un ensemble de feuilles qui contiennent l'expression des propriétés sémantiques caractérisant la classe à laquelle appartient l'objet atteint à travers la chaîne analysée. L'intervention de la grammaire opérationnelle définit sur cette expression terminale de la grammaire conceptuelle, dans le contexte de l'objet considéré, un algorithme d'exécution d'actions primitives. Cet algorithme inclut, en particulier, le contrôle de la possibilité d'exécution des actions primitives requises, pour la classe d'objets mise en jeu, dont certaines sont exécutées

dès cette phase. Il détermine le schéma sémantique qui permet ensuite la génération du code correspondant. Nous indiquerons au paragraphe 4.2 les schémas sémantiques correspondant à la création de tous les objets dont les classes sont engendrées par la grammaire conceptuelle du langage de base.

3.4 - LES VALEURS EXTERNES -

Dans tous les langages sont nécessaires des instructions d'entrée-sortie, qui font intervenir des valeurs externes, c'est-à-dire des valeurs, exprimées dans la syntaxe du langage, mais sur un support externe à l'ordinateur. Dans un modèle de formalisation tel que celui que nous introduisons, ces valeurs définissent une classe d'objets, qu'il ne nous a pas semblé utile de décrire de manière formelle, comme les classes citées au paragraphe 3.2 et engendrées par la grammaire conceptuelle exposée au paragraphe 3.3. Nous indiquons néanmoins très grossièrement les propriétés caractérisant cette classe, afin que le lecteur comprenne, en particulier l'intérêt et la signification des valeurs de type format, introduites dans ce chapitre.

Une valeur externe est définie par l'ensemble des trois propriétés suivantes :

- une localisation : cette action, dont le rôle est identique à une désignation de valeur interne, permet l'accès à la valeur externe : elle consiste en la lecture des paramètres d'état du support externe avec lequel se fait l'échange de la valeur, au moment de l'élaboration de l'instruction d'échange.

Il y a deux types de localisation : la première, appelée mise en page, concerne les échanges de valeur de l'ordinateur vers le milieu extérieur ; la seconde, appelée prise en page, concerne les échanges de valeurs du milieu extérieur vers l'ordinateur,

- une lecture-f, qui spécifie, par la considération d'un objet de type constante ou variable de format, la présentation et le type de la valeur échangée,

- une lecture, ou une écriture, dont l'exécution réalise l'échange de valeur.

Cet ensemble de propriétés définit deux types de valeurs externes : les valeurs en entrée, les valeurs en sortie.

Une valeur en entrée est caractérisée par l'ensemble de propriétés suivantes :

- une prise en page,
- une lecture-f,
- une lecture.

Une valeur en sortie est caractérisée par l'ensemble de propriétés suivantes :

- une mise en page,
- une lecture-f,
- une écriture.

L'exécution de ces trois actions définit l'algorithme de création d'une valeur externe : la durée de vie de cette valeur est le temps d'élaboration de l'instruction d'échange qui l'a créée.

CONCLUSION -

Ce modèle de définition formelle des langages, présenté de manière très schématique, est davantage développé dans les chapitres suivants où il est appliqué à la définition partielle du langage de base et à la définition du processus d'extensibilité : nous verrons alors, comment cette définition, très proche d'une description formelle de l'implémentation des objets et des traitements considérés, constitue une définition extensible des langages.

CHAPITRE 4

LE LANGAGE DE BASE

CHAPITRE 4

LE LANGAGE DE BASE

INTRODUCTION -

On appelle langage de base, le langage à partir duquel on initialise tout processus de génération d'un langage par extensibilité : ce langage est un langage cohérent, c'est-à-dire que son compilateur donne accès à tous les objets qui y sont définis et prévoit leurs traitements éventuels.

Les symboles de base de ce langage, c'est-à-dire les éléments terminaux syntaxiques des grammaires qui l'engendrent, réalisent une représentation dite canonique de tous les langages engendrés par extensibilité. Lorsqu'on atteindra ainsi un langage classique, il suffira de faire une transcription des symboles pour le retrouver dans sa représentation habituelle.

Les phrases du langage de base expriment l'accès à différentes catégories d'objets et à leurs possibilités de traitements : tel qu'il est exposé dans ce chapitre, ces objets et traitements n'ont pas nécessairement le caractère primitif qui semble, à priori, être requis pour une définition de langage de base. Aussi nous verrons que le langage proposé pour langage de base est retenu provisoirement, mais son choix actuel est justifié parce que le langage de base offre ainsi deux qualités :

- son aptitude à décrire convenablement et simplement le mécanisme d'extensibilité exposé au chapitre 5,

- l'accès qu'il exprime à des catégories d'objets, ainsi qu'à leur traitement qui en fait un langage cohérent dans lequel on pourra programmer à l'aide de la représentation canonique.

Au point de vue catégorie d'objets, le langage de base en distingue deux, que nous appellerons, pour respecter une dénomination plus classique, les valeurs et les instructions.

Les instructions ne sont pas traitées dans ce travail, mais on peut d'ores et déjà imaginer que le langage de base permettra, dans un premier temps, et pour les raisons exposées ci-dessus, sans trop se soucier de l'aspect primitif de ces instructions, l'utilisation des instructions classiques des langages évolués tels que FORTRAN ou ALGØL 60, et leur séquençement.

Les valeurs traitées dans ce travail sous leur aspect conceptuel, défini en termes de classes d'objets, à l'aide de diverses actions primitives décrites au chapitre 3, sont les suivantes :

- les valeurs simples : constantes, variables ou pointeurs des langages de type ALGØL,
- les valeurs composées, constantes ou variables, correspondant aux tableaux, éventuellement flexibles, et structures de certains langages de type ALGØL,
- les valeurs dites formelles, simples ou composées, dont on peut rapprocher l'utilisation de celle des chaînes littérales d'APL [De R 74],
- les valeurs format, simples ou composées, qui sont des paramètres permettant le transfert de valeurs,
- les valeurs de type notion, dont le rôle est l'objet du chapitre 5.

L'aspect de ces valeurs engendré par la grammaire conceptuelle du langage de base est exposé au paragraphe 4.1. Des exemples, où sont illustrées la syntaxe et la sémantique de ces objets, telles qu'elles sont introduites par cette grammaire, sont données au paragraphe 4.2.

Au point de vue traitements, exposés de manière informelle au paragraphe 4.3, ils concernent la définition et l'utilisation des différents objets dont les classes sont définies ci-dessus : nous étudierons principalement la déclaration et l'effectivation (*).

(*) néologisme signifiant la transformation d'une valeur formelle en valeur effective.

4.1 - LA GRAMMAIRE CONCEPTUELLE -

La grammaire conceptuelle du langage de base est de la forme exposée au paragraphe 3.3.1.

Nous noterons les symboles non terminaux en lettres majuscules :

$N = \{\text{CONCEPT, D-ALLOC, DESIGNATION, DOMINANTE, EPRIMITIF, EREFLEX, FORMULATION, GROUPE, GROUPIDE, IDENTIFICATION, NOTATION, NOTYPE, PRIMITIF, REFLEX, S-ALLOC, TYPE, VIDE}\}$

- CONCEPT est l'axiome de cette grammaire.

- DOMINANTE est la notion caractérisant la zone dominante qui spécifie le genre de chaque classe d'objets.

- Les trois façons de désigner une constante sont exprimées par les dérivations de la notion DESIGNATION, qui contiennent une occurrence de la notion IDENTIFICATION, FORMULATION ou NOTATION : ces symboles non terminaux ont des dérivations sur les parties sémantiques desquelles agissent des productions terminales de la grammaire opérationnelle, pour préciser, en particulier la durée de vie d'une valeur, c'est-à-dire la portée de sa déclaration, au sens des langages de type ALGØL : nous ne préciserons pas davantage, dans ce travail, ces dérivations et ces relations entre la grammaire conceptuelle et la grammaire opérationnelle.

- Les dérivations des notions EPRIMITIF et PRIMITIF précisent le type d'objet mis en place par une écriture et n'apparaissent donc qu'une fois dans toute dérivation : ceci signifie qu'il n'y a pas nécessairement, dans le langage de base, indications du type de la valeur déclarée, au sens indication de mode ALGØL 68, et qu'il n'y a donc pas systématiquement contrôle statique de ce type.

- D-ALLOC et S-ALLOC précisent le type d'allocation faisant partie de la définition de la classe d'objets engendrés.

- La dérivation de GROUPE présente un aspect syntaxique et un aspect sémantique : cette dérivation produit un ensemble de zones, appelé zone composée, sur chaque élément duquel est répétée la possibilité d'action primitive, écriture qui apparaît en occurrence la plus à droite derrière l'occurrence de GROUPE, dans l'ordre lexicographique de la partie droite de la règle de production utilisée dans cette dérivation. L'expression qui suit l'occurrence de GROUPE détermine la classe d'un objet obtenu par une sélection.

On remarquera qu'une valeur dont une partie de la syntaxe et de la sémantique dérive de GROUPE ne comporte qu'une et une seule zone composée.

Nous noterons les symboles terminaux syntaxiques en lettres minuscules :

$T = \{\text{ent, format, soit, [soit], rep, flex, !, notion, []}\}$

Nous noterons les symboles terminaux sémantiques, en italiques :

$\Sigma = \{\text{écriture, -v, -a, -f, -d, -a-d-alloc, -d-s-alloc, -n, écrituride, groupe, }\}$

Remarques préliminaires -

1 - Certaines règles, voire certaines notions, sont redondantes, mais sont énoncées, comme pour les grammaires figurant dans les rapports de définition des langages comme ALGØL 60 ou ALGØL 68, dans un but de clarification mnémonique pour le lecteur.

2 - Dans l'ensemble des règles énoncées ci-dessous, la dérivation de certains symboles non terminaux, jusqu'à une suite de symboles terminaux syntaxiques ou sémantiques, n'est pas donnée parce que cette dérivation est liée à certaines productions terminales de la grammaire opérationnelle.

3 - On ne considère, pour simplifier l'exposé, qu'un seul type de valeurs arithmétiques ou logiques : de type entier.

4 - Nous indiquons par le symbole ":" la relation de production : les éventuelles alternatives de la partie droite d'une règle sont séparées par le symbole ";". La fin de règle est indiquée par le symbole ".".

Enoncé des règles -

CØNCEPT : DOMINANTE DESIGNATION.

D-ALLOC : ! GROUPIDE *écriture -a-d-alloc.*

DESIGNATION : IDENTIFICATION ; NOTATION ; FORMULATION.

DOMINANTE : (S-ALLOC GROUPE)^{0|1} *écriture* PRIMITIF ; D-ALLOC *écrivuride* EPRIMITIF.

EPRIMITIF : NOTYPE ; EREFLEX.

EREFLEX : *-a-d-alloc rep ; -d-s-alloc flex groupe écriture -a-d-alloc.*

GRUPE : [] *grupe*.
GROUPIDE : S-ALLOC GRUPE GROUPIDE ; VIDE.
NOTYPE : TYPE ; -*n* notion.
PRIMITIF : NOTYPE ; REFLEX.
REFLEX : -*a* soit ; -*d* [soit].
S-ALLOC : *écriture* -*d-s-alloc*.
TYPE : -*v* ent ; -*f* format.
VIDE : .

4.2 - EXEMPLES -

Nous donnons dans ce paragraphe quelques exemples de déclarations du langage de base. Ces exemples n'ont pas pour but d'illustrer l'algorithme de déclaration, tel qu'il a été présenté, de manière informelle, au chapitre précédent. On dégage à chaque exemple l'ensemble des propriétés conceptuelles qui caractérisent la classe de l'objet créé par l'élaboration de la déclaration considérée : cet objet syntaxiquement exprimé à gauche du symbole "=" de déclaration, constitue ce qu'on appelle généralement la partie gauche de la déclaration, l'autre partie constituant sa partie droite.

Nous ne donnons pas d'exemple de valeur de classe notion, dont l'étude est reportée au chapitre 5.

Toutes les valeurs présentées sont de classe []^{0|1} PRIMITIF
ou ! []* EPRIMITIF,
où PRIMITIF engendre les constantes simples : effectives ou formelles,
EPRIMITIF engendre les constantes simples effectives,
[] est le symbole syntaxique des valeurs composées,
et ! est le symbole syntaxique des constantes d'adresse.

On trouvera pour tous les exemples, la déclaration elle-même, son objet, l'ensemble des propriétés sémantiques de la valeur créée, ainsi que les observations que celles-ci appellent, la dérivation dans la grammaire conceptuelle qui a permis d'engendrer la classe considérée, puis la représentation physique du schéma sémantique ainsi défini, dans un modèle d'implantation du type de celui présenté en [Gr A 75] : nous rappelons que ces schémas

expriment, à travers ce modèle d'implémentation, un ensemble de rôles sémantiques : ils ne préjugent pas d'une implantation mémoire réelle des objets étudiés, pour laquelle une optimisation est naturellement possible.

4.2.1. - Les constantes simples -

4.2.1.a - déclaration : `enti = 3`

objet : déclaration explicite d'une constante de valeur identifiée par `i`.

propriétés : après élaboration de cette déclaration, `i` désigne un objet placé dans une zone dominante, et dont la dénotation est 3.

dérivation dans la grammaire conceptuelle :

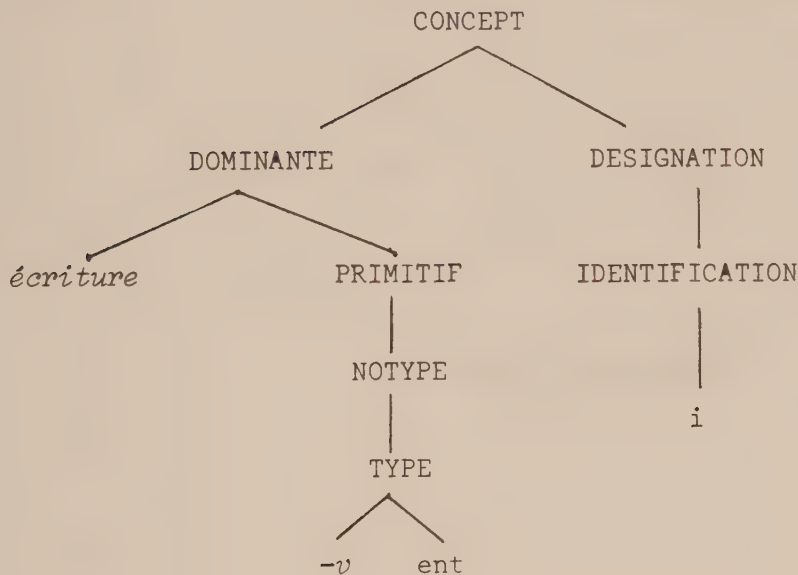
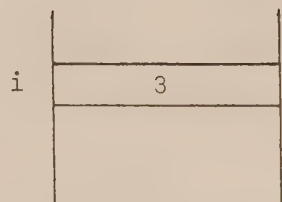


schéma sémantique :



pile d'identificateurs

4.2.1.b - déclaration : format x = \$...\$

objet : déclaration explicite d'une constante de format identifiée par x.

propriétés : l'élaboration de cette déclaration consiste en l'écriture-f de la dénotation de constante de format implicitement déclarée en partie droite.

dérivation dans la grammaire conceptuelle :

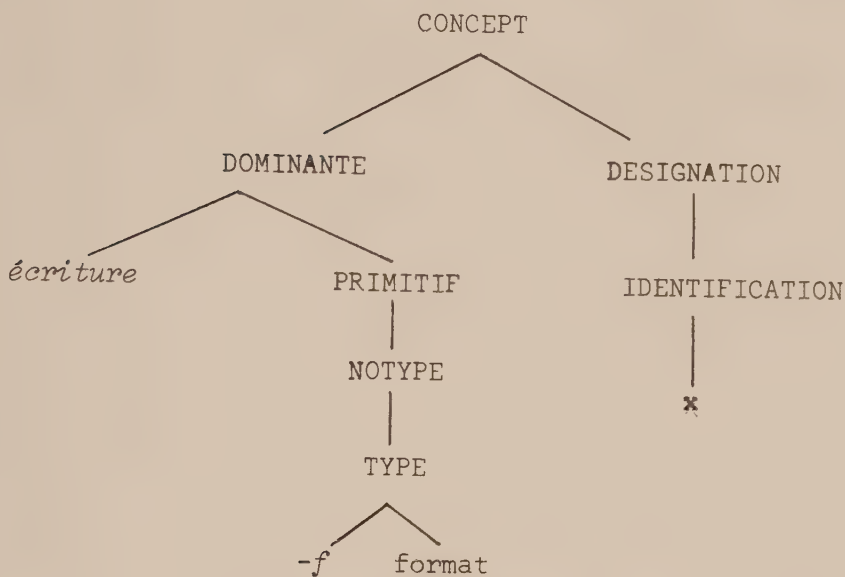
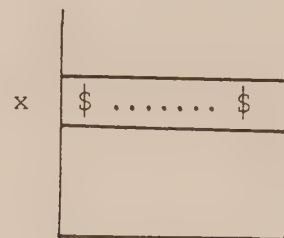


schéma sémantique :



pile d'identificateurs

4.2.1.c - déclaration : soit x = vide

objet : déclaration explicite d'une constante formelle identifiée par x.

propriétés : x désigne une zone dominante, élément de la table des valeurs formelles. Après élaboration de la déclara-

tion, le contenu de la zone est une codification de x.

dérivation dans la grammaire conceptuelle :

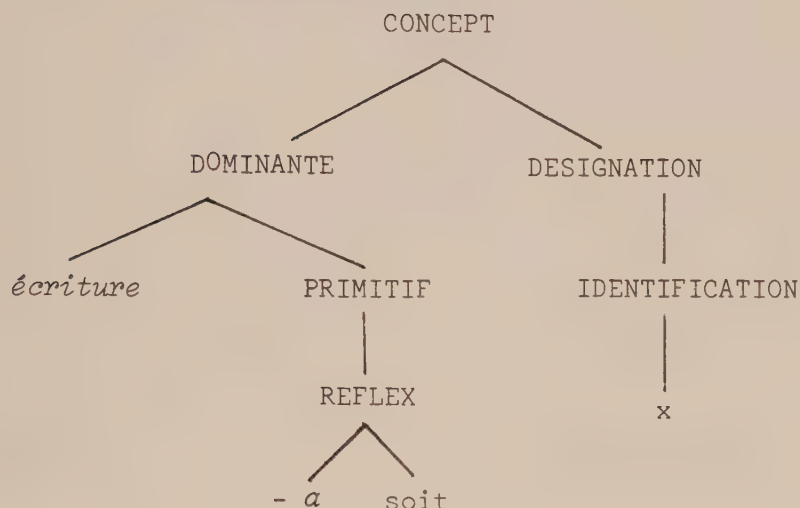


schéma sémantique :

x	
	Codification

table des valeurs formelles.

remarque : Une des utilisations essentielles de telles valeurs est la possibilité d'écriture de fonctions de type fonctionnelles telles que des fonctions de dérivation ou d'intégration. On peut sans doute, en s'inspirant des chaînes littérales du langage APL appréhender d'autres utilisations de ces constantes [De R 74].

4.2.1.d - déclaration : [soit] x = [vide]

objet : déclaration explicite d'une constante composée formelle identifiée par x.

propriétés : x désigne une zone dominante, élément de la table des valeurs formelles ; après élaboration de la déclaration, le contenu de la zone est :

- une codification de x,
- un descripteur formel.

dérivation dans la grammaire conceptuelle :

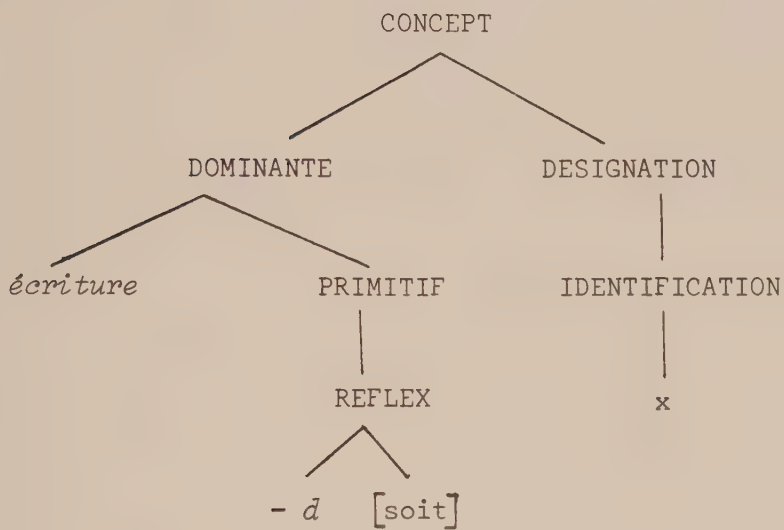


schéma sémantique :

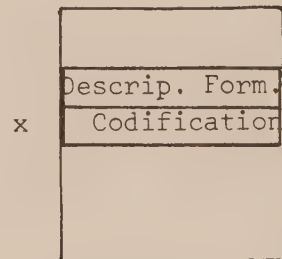


table des valeurs formelles.

4.2.2 - Les constantes composées -

4.2.2.a - déclaration : [] ent t = (1,2,3)

objet : déclaration explicite d'une constante composée de valeur, identifiée par t.

propriétés : t désigne une zone dominante de la pile d'identificateurs sur laquelle est exécutée l'écriture-d du descripteur fictif de la constante multiple dénotée (1,2,3), dont la déclaration est implicite. A cette zone est allouée statiquement un ensemble de zones dominantes, applé zone dominante composée, dans la partie dynamique de la pile d'identificateurs. La remarque sur la notion GROUPE, énoncée au paragraphe 4.1 précise que chacune de ces zones dominantes est une constante de valeur.

dérivation dans la grammaire conceptuelle :

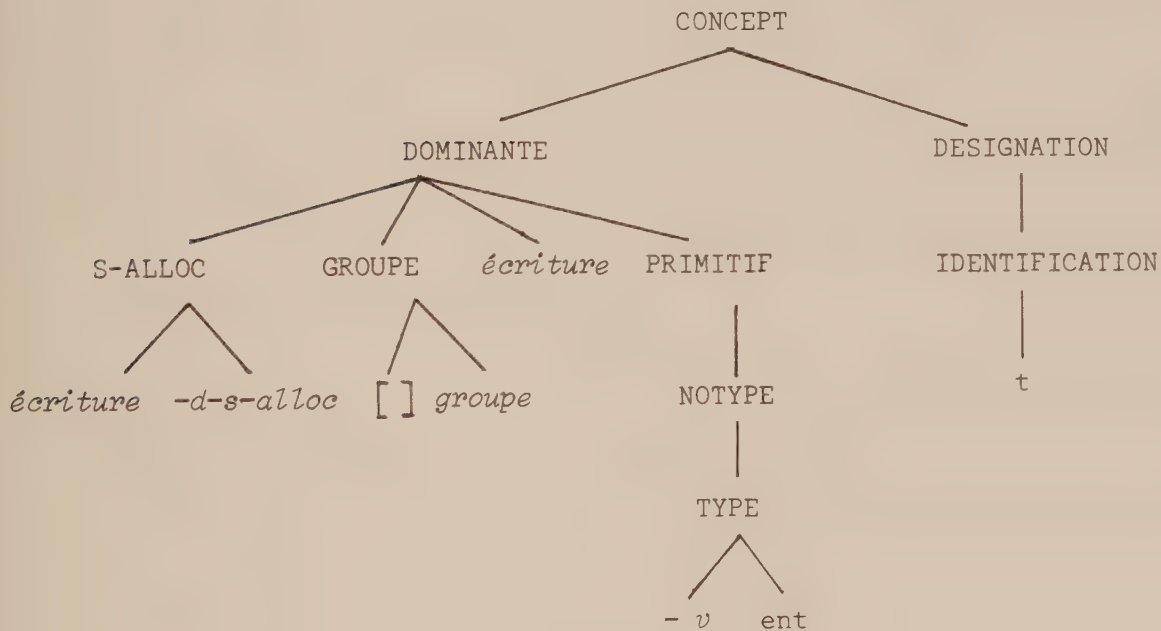
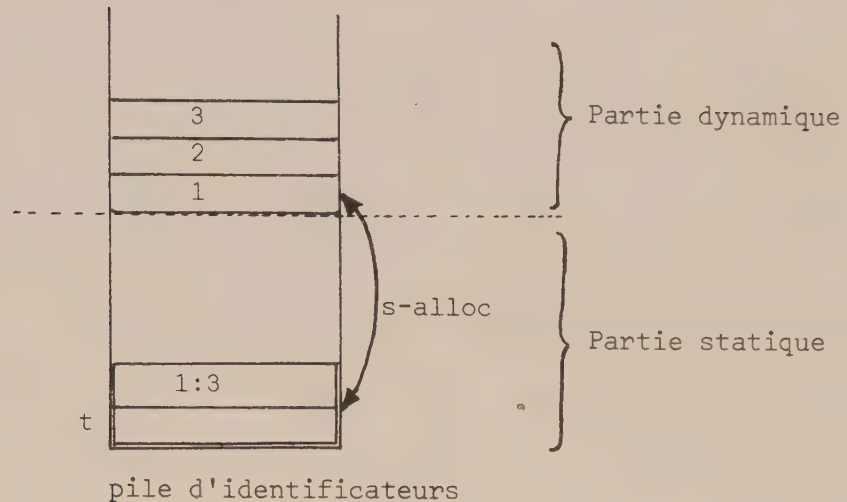


schéma sémantique :



remarque : il n'y a pas de possibilité d'affectation globale à la valeur, ni à chacun des éléments puisque l'unique faculté d'écriture est épuisée à la déclaration. Cette remarque est également valable pour les trois déclarations suivantes.

4.2.2.b - déclaration : [] format $t = x$, où x est une constante composée de format précédemment déclarée.

objet : déclaration explicite d'une constante composée de format, identifiée par t .

propriétés : id. à la précédente
Chaque zone dominante constituant la zone composée est une constante de format.

dérivation dans la grammaire conceptuelle :

Cf. la précédente.

schéma sémantique :

Cf. le précédent.

4.2.2.c - déclaration : [] soit $x = (i, j, k)$
où i, j, k sont des constantes formelles précédemment déclarées.

objet : déclaration explicite d'une constante composée de valeurs formelles, identifiée par x.

propriétés : x désigne une zone dominante de la pile d'identificateurs, sur laquelle est exécutée l'écriture-d du **descripteur** de la partie droite. A cette zone est allouée statiquement une zone dominante composée, dans la table des valeurs formelles. Chacune des zones dominantes qui la compose est une constante formelle.

dérivation dans la grammaire conceptuelle :

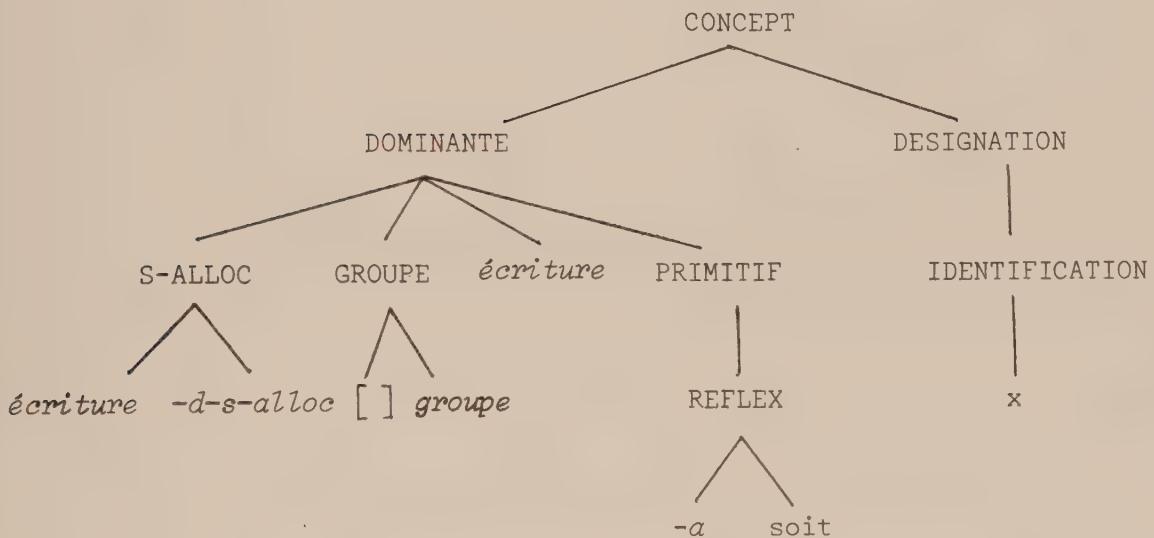
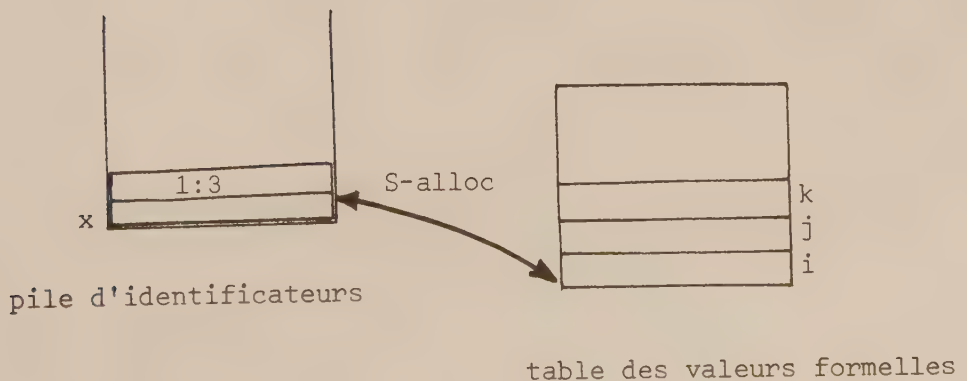


schéma sémantique :



4.2.2.d - déclaration : [] [soit] x = [1:3] [vide]

objet : déclaration explicite d'une constante composée de constantes composées formelles, identifiée par x.

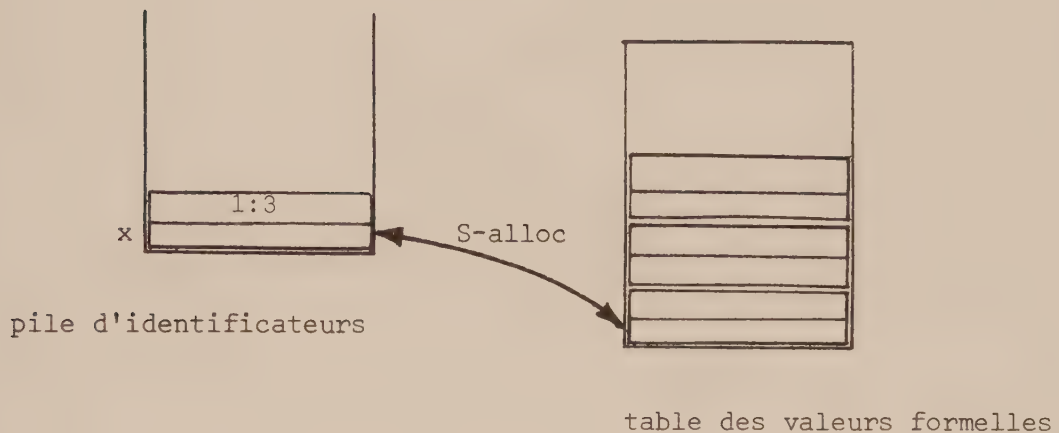
propriétés : id. à la précédente.

Chaque zone dominante constituant la zone composée est une constante composée formelle.

dérivation dans la grammaire conceptuelle :

Cf. la précédente.

schéma sémantique :



4.2.3 - Les constantes d'adresse simple -

Une constante d'adresse possède, parmi ses propriétés sémantiques celle de l'allocation dynamique d'une zone dominée, à l'élaboration de la déclaration : on réalise alors une variable de genre, où genre spécifie le type d'écriture qui peut être exécutée, un nombre arbitraire de fois sur la zone dominée : nous définissons successivement les variables de valeur, les variables de format, les variables d'adresse et les variables de descripteur.

Nous verrons dans les exemples qui suivent les diverses déclarations possibles, implicites ou explicites, qui peuvent conduire à une allocation dynamique ou d'une zone dominée "vierge", ou d'une zone déjà allouée à une autre constante d'adresse (synonymie au sens ALGØL 68).

4.2.3.a - déclaration : ! ent i = loc

objet : déclaration explicite d'une constante d'adresse identifiée par i et réalisation d'une variable de valeur.

propriétés : à la zone dominante est allouée dynamiquement par le générateur local d'emplacement : loc, une zone dominée dans la pile des générateurs locaux, sur laquelle pourront être exécutées un nombre arbitraire de fois des écritures-v. L'élaboration de cette déclaration réalise une variable de valeurs.

dérivation dans la grammaire conceptuelle :

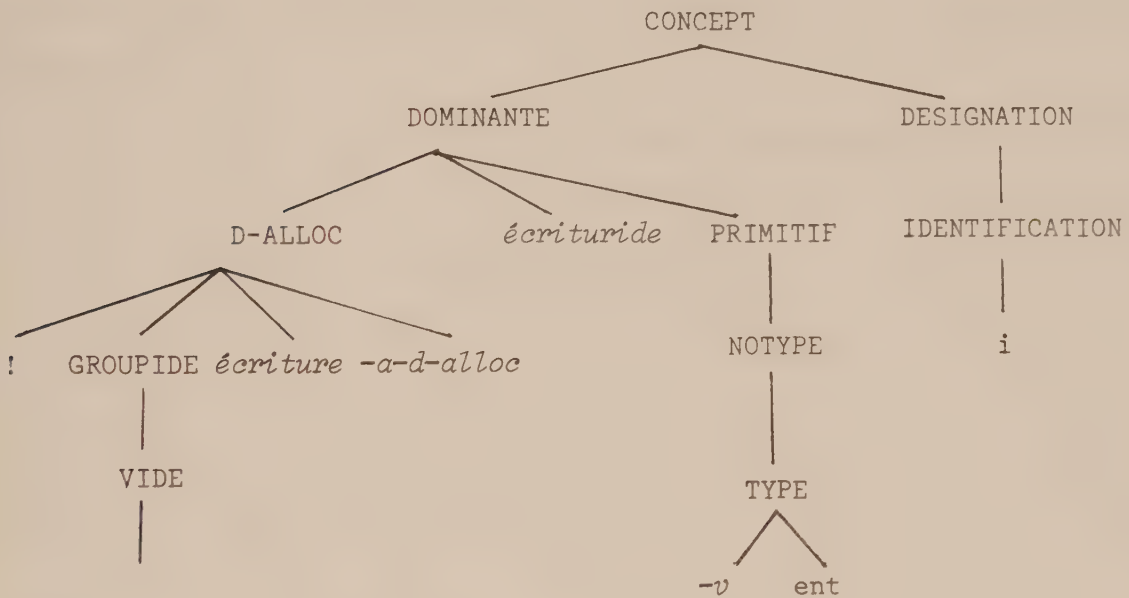
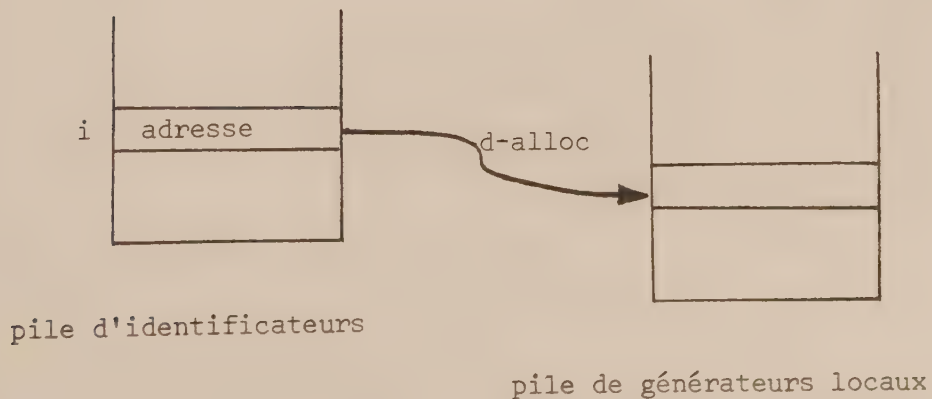


schéma sémantique :



remarque : il existe également un générateur global d'emplacement : tas, qui alloue une zone dominée dans le tas.

4.2.3.b - déclaration : ! format t = f format 3

objet : déclaration explicite d'une constante d'adresse identifiée par t et réalisation d'une variable de format.

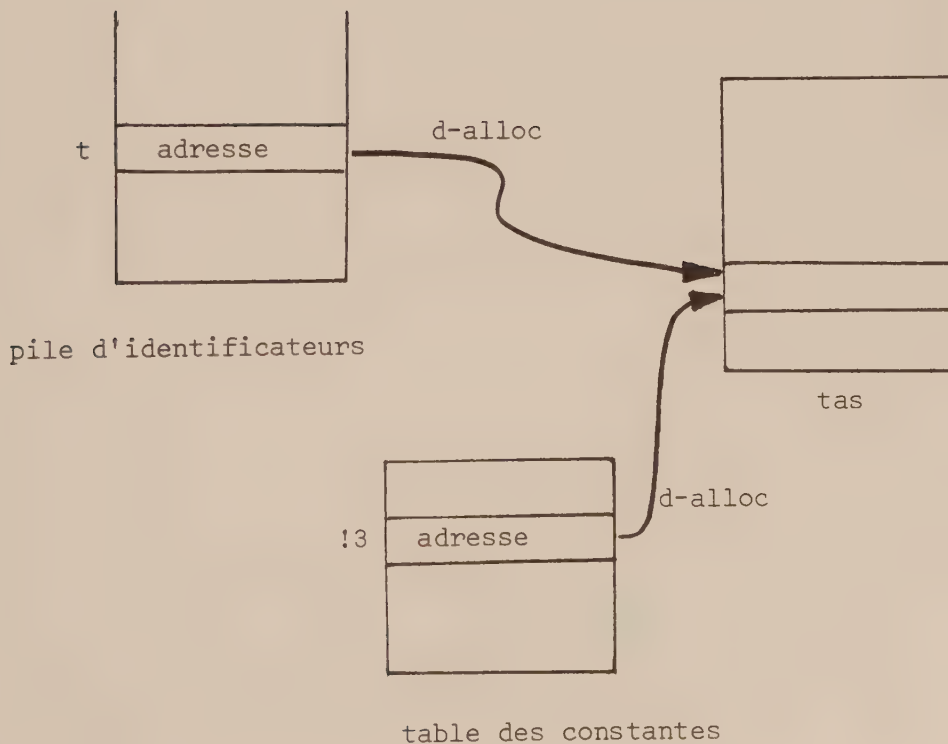
propriétés : la partie droite de la déclaration est une déclaration implicite de constante d'adresse, dénotée par 3. L'allocation dynamique est faite avec un emplacement du tas, pour réaliser une variable de format.

A la zone dominante associée à t est allouée dynamiquement la même zone dominée. On réalise ainsi une variable de format synonyme de la précédente.

dérivation dans la grammaire conceptuelle :

C5- la précédente.

schéma sémantique :



Remarque : alors que la zone dominée est adressable par l'intermédiaire de la dénotation !3 durant tout le programme, elle ne l'est par l'intermédiaire de t que dans une certaine partie de ce programme.

4.2.3.c - déclaration : ! rep $x = y$

où y est une variable d'adresse précédemment réalisée.

objet : déclaration explicite d'une constante d'adresse, identifiée par x , et réalisation d'une variable d'adresse.

propriétés : à la zone dominante de x est allouée dynamiquement la zone dominée de la variable y , sur laquelle peuvent être exécutées des écritures-a. On réalise une variable d'adresse, ou pointeur dynamique, x synonyme de y .

dérivation dans la grammaire conceptuelle :

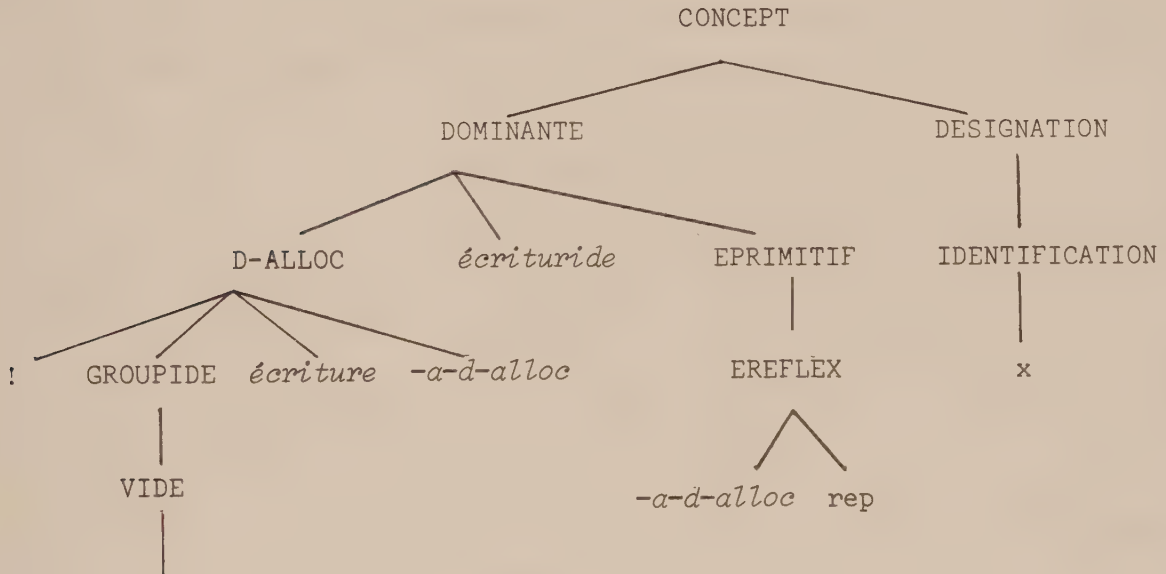
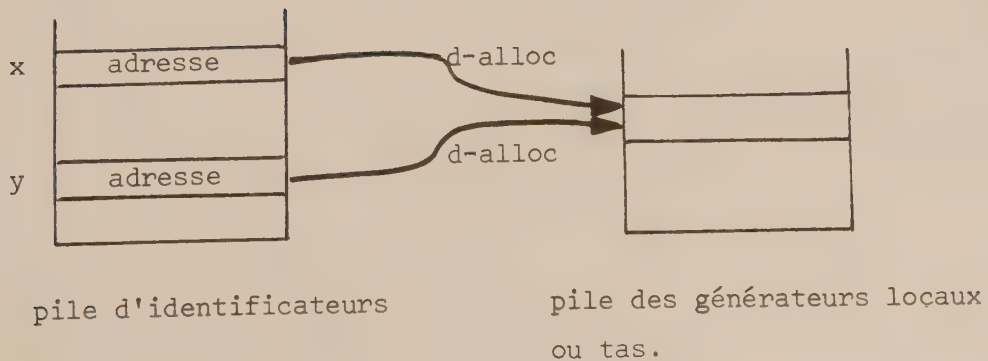


schéma sémantique :



Remarque : On voit ici apparaître la distinction essentielle, c'est-à-dire conceptuelle, des symboles de base ! et rep, confondus en ALGØL 68 en le même symbole terminal rep : si tous deux correspondent à des écritures-a, l'un les réalise sur une zone dominante pour définir les constantes d'adresses, l'autre les réalise sur une zone dominée.

Le premier permet plusieurs accès indépendants à la même information, sans qu'il soit utile d'en considérer plusieurs copies.

La confusion introduite dans ALGØL 68 entre ces deux niveaux de rep, dénoncée par LEROY dans [Le 72], comme une confusion conceptuelle entre référence indirecte et variable, conduit, sans aucun doute à l'absence du minimum d'élégance, de rigueur conceptuelle, de clarté et de généralité requises pour la description d'un langage de haut niveau.

4.2.3.d - déclaration : ! flex x = loc

objet : déclaration explicite d'une constante d'adresse identifiée par x et réalisation d'une variable de descripteur.

propriétés : à l'élaboration de la déclaration, une écriture-a est exécutée sur la zone dominante, qui lui alloue dynamiquement une zone dominée sur laquelle pourront être exécutées des écritures-d. Le générateur d'emplacement situé en partie droite spécifie la localisation de la zone dominée : elle est un emplacement de la pile des générateurs locaux si le générateur est 'loc' ; elle est un emplacement du tas si le générateur est 'tas'. Une autre zone dominante de la pile d'identificateurs est allouée statiquement : sur cette zone peut être exécutée une écriture-a qui lui alloue dynamiquement une zone dominée composée, placée dans le tas. L'élaboration de cette déclaration réalise une variable de descripteur, ou tableau dynamique.

dérivation dans la grammaire conceptuelle :

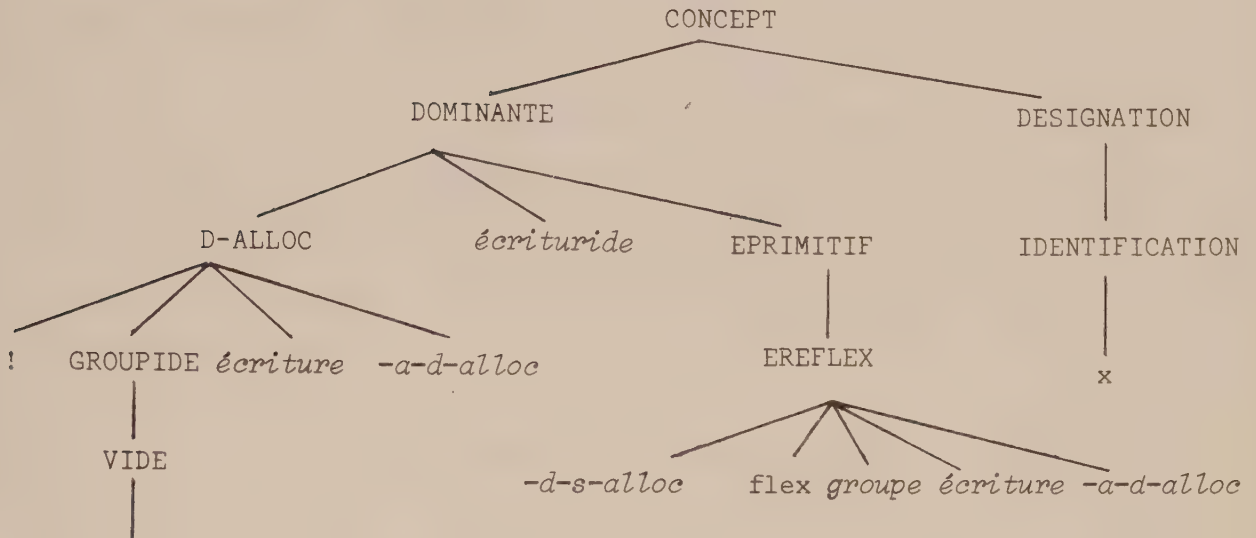
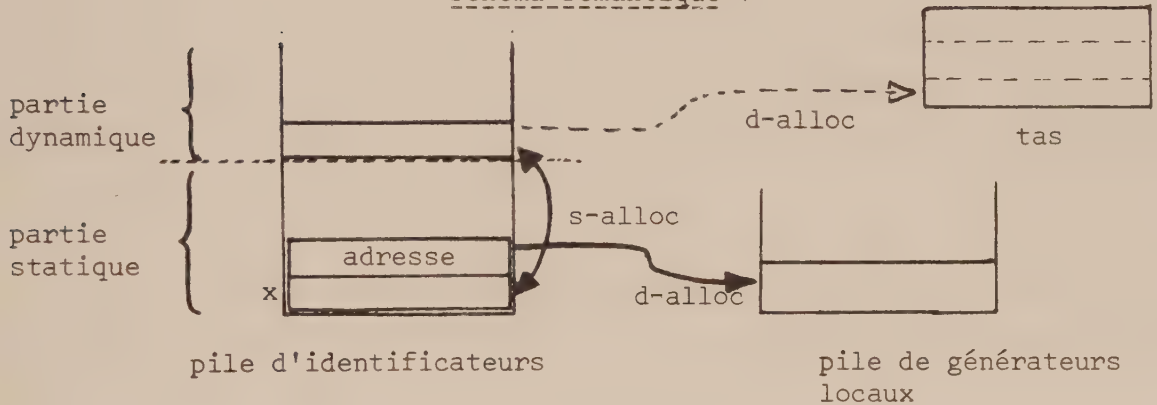


schéma sémantique :



remarques :

1/ L'action primitive écriture-a sur la seconde zone dominante n'est pas exécutée lors de la déclaration.

Lés éléments du tableau ne sont qu'à l'état de classe de valeur, sous forme de constante d'adresse : ils ne seront déclarés, implicitement, que lors d'une première affectation, qui exécute l'action primitive écriture-a.

2/ La taille de la zone dominée composée n'est connue qu'à l'exécution et est variable par affectation : les paramètres de fonction d'accès sont en effet variable par écriture-d. De ce fait la zone dominée composée est placée dans le tas, et son allocation dynamique

à la zone dominante est réalisée par l'écriture-a d'une adresse, susceptible d'être changée par des procédures de gestion mémoire, du type ramasse-miettes, ignorées du programmeur.

3/ De tels tableaux dynamiques réalisent, mise à part une indication dynamique du mode des éléments, les tableaux flexibles ALGØL 68. L'implantation mémoire proposée ici à travers le schéma sémantique nous semble convenir moyennant optimisation, à ces tableaux ALGØL 68, que bien des implémentations actuelles confondent avec les noms de noms de tableaux.

4.2.4. - Les constantes d'adresse composée -

Une constante d'adresse composée possède, parmi ses propriétés sémantiques, celle de l'allocation statique d'une zone dominante : à l'élaboration de la déclaration est exécutée sur cette zone une écriture-a qui lui alloue dynamiquement un ensemble, de zones dominées, appelé zone dominée composée, pour réaliser une zone composée de genre, où genre spécifie le type d'écriture qui peut être exécutée, un nombre arbitraire de fois sur chacune des zones dominées.

Nous définissons successivement les variables composées de valeurs, les variables composées de formats, les variables composées d'adresses et les variables composées de descripteurs.

Remarque_1 -

La zone dominante à laquelle est allouée dynamiquement la zone dominée composée permet l'affectation globale à la variable composée réalisée.

Remarque_2 -

La sélection d'un élément d'une variable composée détermine un objet de classe variable : on peut donc faire une affectation à chaque élément d'une variable composée.

4.2.4.a - déclaration : $![]$ ent $t = [1:n]$ loc

objet : déclaration explicite d'une constante d'adresse composée identifiée par t et réalisation d'une variable composée de valeurs.

propriétés : la zone dominée composée allouée dynamiquement est une zone de la pile des générateurs locaux. Lorsque la partie droite de la déclaration contient le générateur global d'emplacement 'tas', cette zone composée est un ensemble d'emplacements du tas. Chaque élément de la valeur composée est une variable de valeur.

dérivation dans la grammaire conceptuelle :

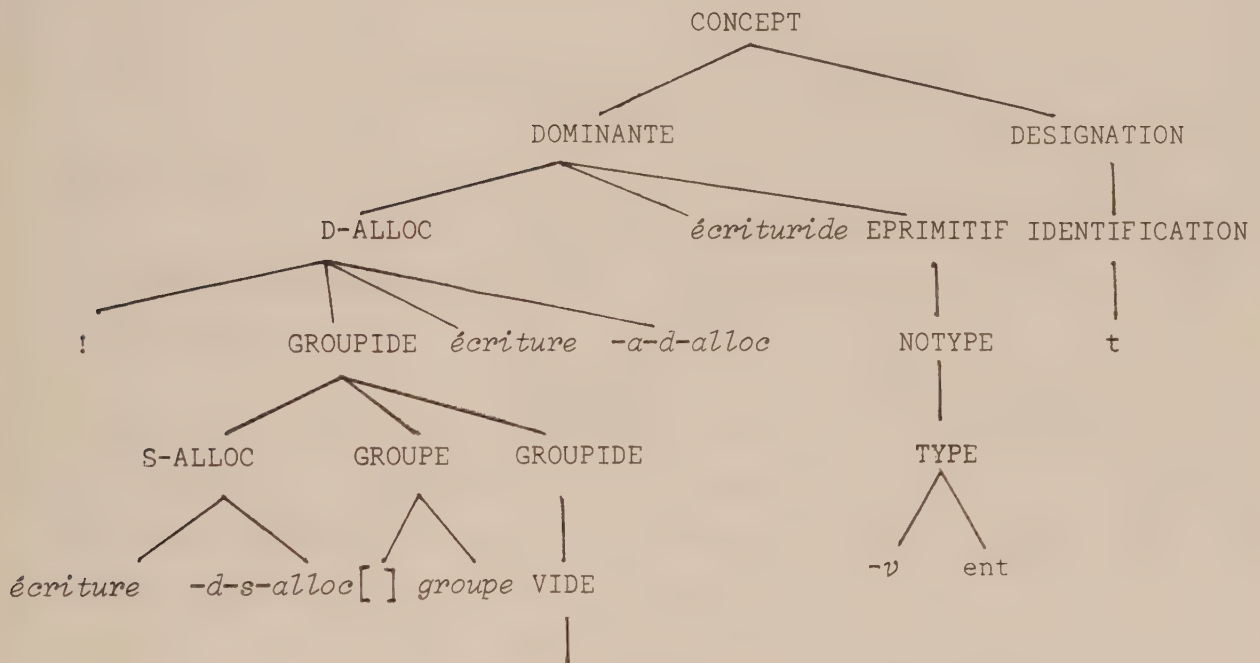
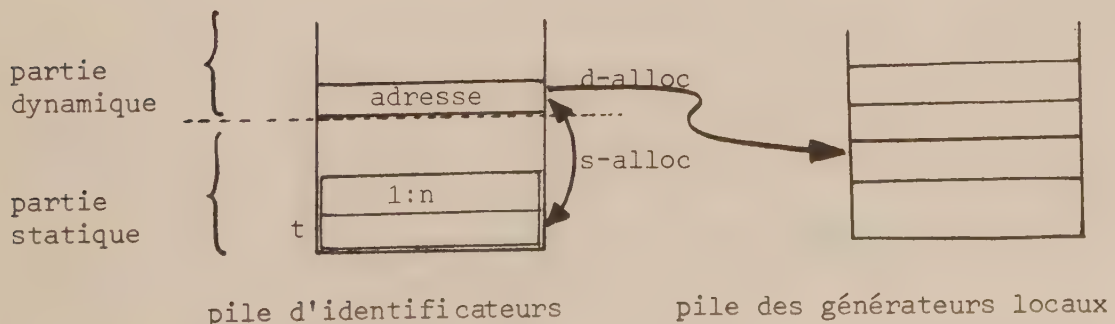


schéma sémantique :



4.2.4.b - déclaration : ! [] format t = [1:n] loc

objet : déclaration explicite d'une constante d'adresse composée identifiée par t et réalisation d'une variable composée de format.

propriétés : id. à la précédente.

Chaque élément de la valeur composée est une variable de format.

dérivation dans la grammaire conceptuelle :

cf. la précédente.

schéma sémantique : id. au précédent.

4.2.4.c - déclaration : ! [] rep t = u

où u est une variable composée précédemment réalisée.

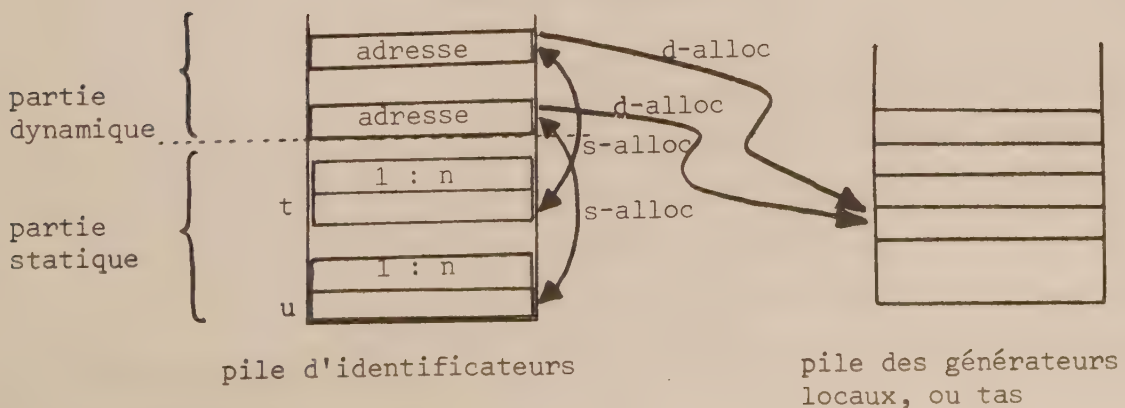
objet : déclaration explicite d'une constante d'adresse composée identifiée par t, et réalisation d'une variable composée d'adresse.

propriétés : la zone dominée composée allouée dynamiquement n'est pas comme dans les deux exemples ci-dessus une zone "vierge" : elle est déjà la zone composée d'une variable u. La variable composée réalisée t est alors synonyme de u. Chaque élément de la valeur composée est une variable d'adresse.

dérivation dans la grammaire conceptuelle :

cf. précédente.

schéma sémantique :



4.2.4.d - déclaration : ![] flex t = [1:n] loc

objet : déclaration explicite d'une constante d'adresse composée identifiée par t et réalisation d'une variable composée de descripteurs.

propriétés : à la zone dominante allouée statiquement est allouée, par l'élaboration de cette déclaration :

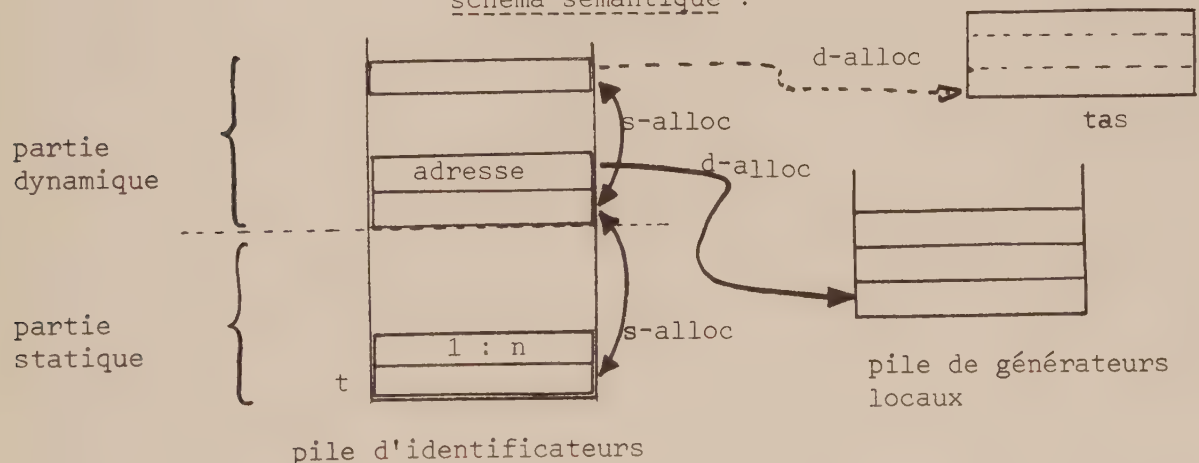
- dynamiquement une zone dominée composée, placée dans la pile des générateurs locaux (pour l'exemple proposé), sur chaque emplacement de laquelle sont exécutables des écritures-d,

- statiquement une zone dominante à laquelle une écriture-a alloue dynamiquement une zone dominée composée, placée dans le tas.

Chaque élément de cette valeur composée est une variable de descripteur à laquelle s'appliquent naturellement les remarques 1 et 2 du paragraphe 4.2.3.d.

dérivation dans la grammaire conceptuelle :
Cf. précédente.

schéma sémantique :



4.2.4.e - Toutes les variables réalisées par l'élaboration des quatre déclarations citées ci-dessus sont des variables composées de genre. Ce sont, pour $n = 1$, des valeurs (variables composées de)ⁿ genre, qui sont pour tout entier n , des valeurs du langage de base.

A titre d'exemple nous donnons une déclaration de constante d'adresse composée dont l'élaboration réalise une (variable composée de)² genre.

déclaration : ! [] [] ent t = [1:n] [1:m] loc

objet : déclaration explicite d'une constante d'adresse composée identifiée par t, et réalisation d'une variable composée de variable composée de valeurs.

propriétés : une telle valeur est assimilable à une variable composée de descripteurs, où l'ensemble des descripteurs variables serait fixé statiquement par une même écriture-d.

Une zone dominante est donc statiquement allouée à la zone dominante de t, par une écriture-d : à cette zone est allouée statiquement une troisième zone dominante, par une écriture-d sur la deuxième. A cette dernière l'exécution d'une écriture-a alloue, dynamiquement, une zone dominée composée, sur chaque élément de laquelle pourront être exécutées des écritures-v.

Chaque élément de la valeur composée ainsi réalisée est une variable composée de valeurs.

dérivation dans la grammaire conceptuelle :

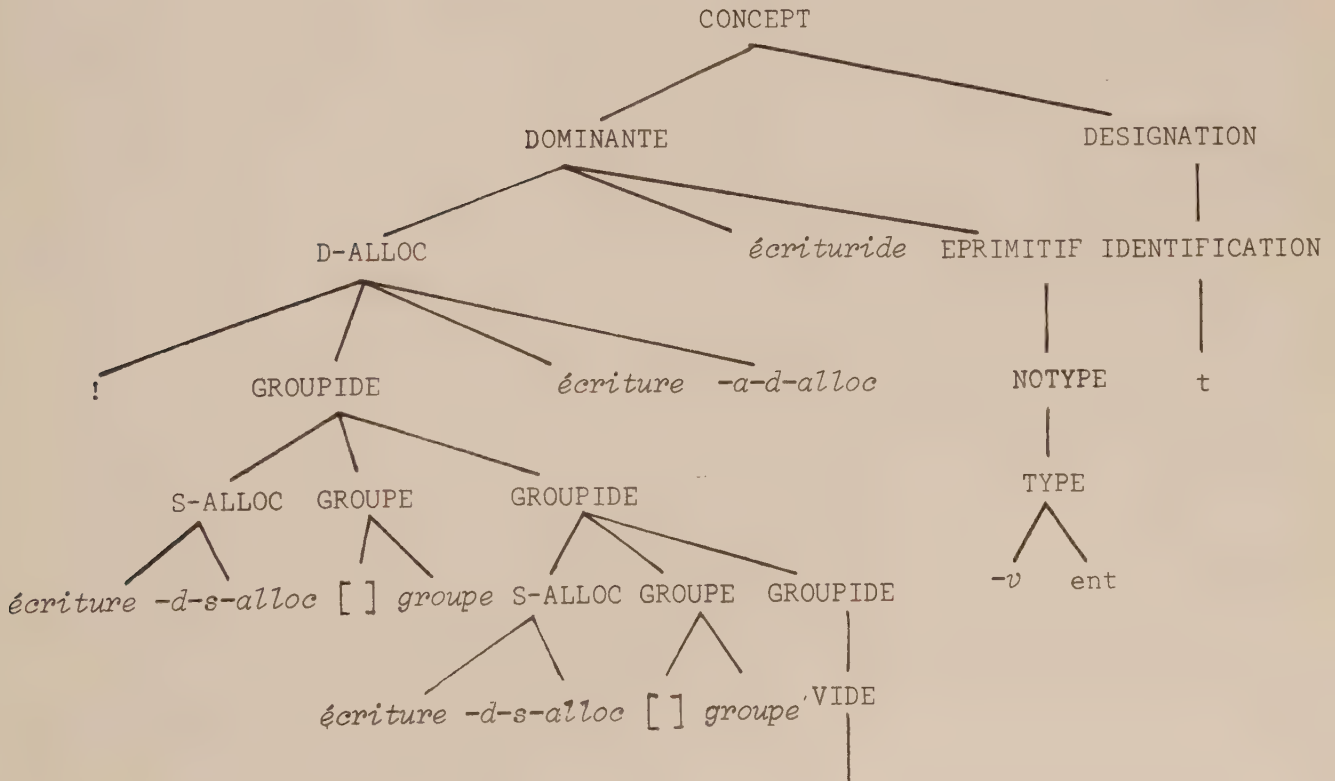
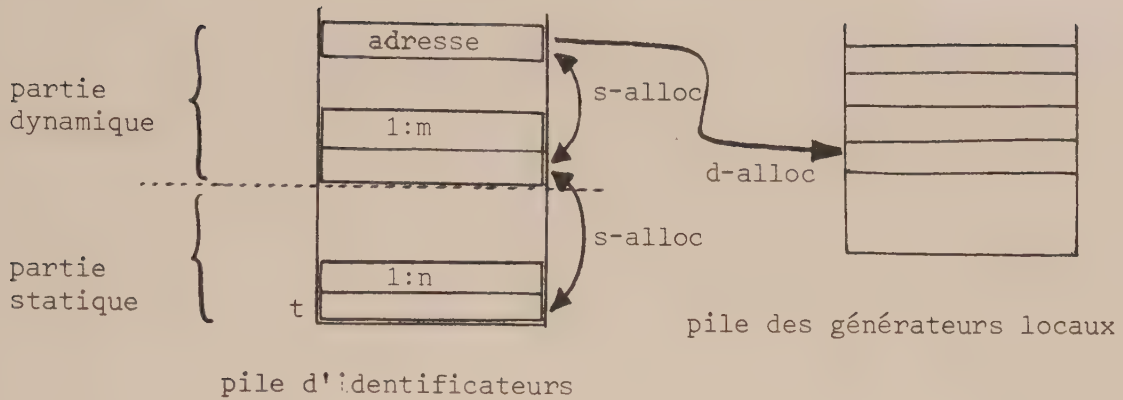


schéma sémantique :



4.3 - LA GRAMMAIRE OPERATIONNELLE -

Nous présentons, de manière informelle, la grammaire opérationnelle du langage de base, dont le rôle est celui exposé dans le modèle de formalisation du chapitre 3.

Outre les opérateurs permettant la définition de valeurs composées et leur accès, opérateurs entrevus au paragraphe 4.1, qui agissent sur la partie sémantique des productions terminales de la notion GROUPE de la grammaire conceptuelle, le lecteur peut admettre que **sont** engendrés par la grammaire opérationnelle, des algorithmes d'exécution de certaines actions primitives, de type : "affectation", "sélection", "effectivation", et naturellement "déclaration" qui assure la création d'un objet.

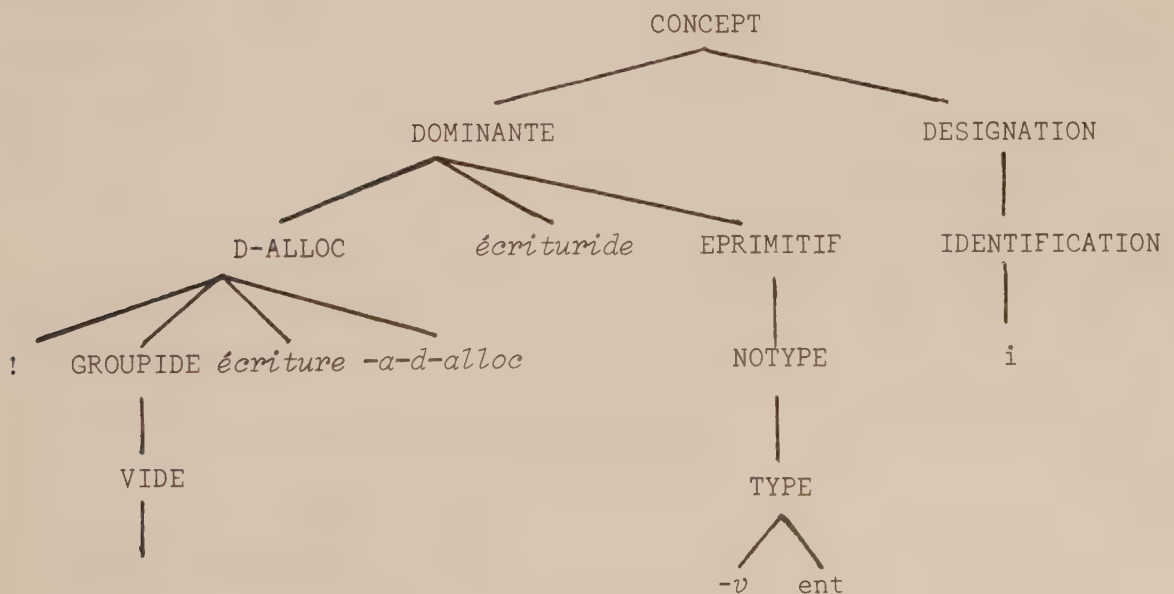
Toutes ces productions de la grammaire opérationnelle agissent comme opérateurs sur la partie sémantique des productions terminales de la grammaire conceptuelle : ainsi l'opérateur "affectation" assure la liaison entre une action primitive lecture et une action primitive écriture de même genre, exécutable sur une zone dominée. Egalement l'opérateur "sélection" qui assure, à partir de l'action primitive lecture-d, la création d'une valeur à partir d'une valeur composée. Nous étudions plus en détail les opérateurs "déclaration" et "effectivation".

4.3.1 - L'opérateur "déclaration" -

Comme il est dit au chapitre 3, l'opérateur "déclaration" crée un objet, par l'exécution d'un ensemble d'actions primitives, parmi celles caractérisant la classe à laquelle il appartient, selon un algorithme qui délivre l'ensemble des informations concernant son implantation en mémoire. Les actions primitives qui ne sont pas exécutées par cet algorithme, restent dans le concept de l'objet créé pour être prises en compte par d'autres opérateurs.

Cet algorithme de déclaration comprend toujours l'exécution de l'unique possibilité d'écriture sur la zone dominante, l'exécution de l'action primitive désignation, et éventuellement, une ou plusieurs allocations statiques ou dynamiques de zones, cette partie non déterminée de l'algorithme pouvant être réalisée après vérification de prédicats sur le genre de l'écriture exécutée.

Exemple : On peut représenter par l'arbre suivant, la dérivation dans la grammaire conceptuelle, de la partie gauche d'une déclaration créant une valeur de type constante d'adresse et réalisant une variable de valeur, (Cf. § 4.2.3.a):



Pour l'élaboration de la déclaration, l'intervention de la grammaire opérationnelle sur cette dérivation, consiste en l'exécution de l'algorithme de l'opérateur "déclaration" qui a pour opérandes, les parties sémantiques, de la production terminale de l'axiome CONCEPT, suivantes : *écriture-a-d-alloc* et la partie sémantique de la dérivation de la notion IDENTIFICATION.

Après cette intervention de la grammaire opérationnelle l'ensemble des informations délivrées précisent l'implantation en mémoire de l'objet désigné par *i* : du point de vue conceptuel, les symboles sémantiques *écritureide* et *-v* sont susceptibles de devenir les opérandes d'un autre opérateur, l'affectation, de la grammaire opérationnelle.

4.3.2 - L'opérateur "effectivation" -

Une valeur formelle est susceptible d'une effectivation dont l'effet, au niveau des valeurs, est comparable à celui de l'opérateur 'execute' ou 'unquote' du langage APL [Sa R 73], dont l'opérande est constitué d'une chaîne de caractères : l'effet de cet opérateur est d'élaborer l'expression, dont l'opérande est le texte, et de substituer le résultat final de cette élaboration à cette expression.

Définition : Dans un modèle d'implémentation tel que celui suggéré au paragraphe 4.2, rendre effective une valeur formelle consiste à lui associer une zone dominante de la pile d'identificateurs.

4.3.2.a - Effectivation d'une constante formelle :

Lorsqu'on rend effective une valeur formelle, on associe à la codification, contenu de sa zone dominante, la désignation d'une zone dominante de la pile d'identificateurs.

La constante formelle déclarée par :

soit $x = \text{vide}$

peut être effectivée :

- en constante de valeur, par : $\text{ent} \leftarrow x$,
- ou en constante d'adresse,

réalisant une variable de valeurs, par : ! ent $\leftarrow$ x,
ou réalisant une variable d'adresse, par : ! rep $\leftarrow$ x.

4.3.2.b - Effectivisation d'une constante composée

formelle :

Lorsqu'on rend effective une constante composée formelle on associe la codification et le descripteur formel aux emplacements correspondants d'une zone dominante de la pile d'identificateurs : la constante composée formelle déclarée par :

[soit] x = [vide]

peut être effectivée - en constante composée de valeur, par : [] ent $\leftarrow$ x,
- ou en constante d'adresse composée, réalisant par exemple une variable composée de valeur, par :
! [] ent $\leftarrow$ x,
- ou en une constante d'adresse réalisant une variable de descripteur, par : ! flex $\leftarrow$ x.

4.3.2.c - Remarques :

1 - Après la déclaration [] soit x = (i,j,k) on peut effectuer les éléments pris un par un, qui sont des valeurs formelles :

ent $\leftarrow$ x [1]

! ent $\leftarrow$ x [3]

mais on remarquera que globalement la valeur est effective.

2 - De même après la déclaration [] [soit] x = [1:3] [vide] la valeur x est globalement effective et on ne pourra effectuer que ses éléments :

! flex $\leftarrow$ x [2]

CHAPITRE 5

LE PROCESSUS D'EXTENSIBILITE

CHAPITRE 5

LE PROCESSUS D'EXTENSIBILITE

INTRODUCTION -

Le principe général de l'extensibilité des langages, tel que nous l'avons défini au chapitre 2, est la possibilité pour tout utilisateur d'un tel langage, de définir un nouvel ensemble de catégories d'objets reconnus comme faisant partie de ce langage.

Nous avons vu au chapitre 2 que ce principe est mis en oeuvre, sous forme de mécanismes particuliers, dans les définitions de type interprétatives des langages évolués récents : mais cette réalisation n'est que partielle, et en particulier l'extensibilité dans le domaine de la sémantique, dite dynamique, c'est-à-dire la définition extensible du répertoire de l'interpréteur, ne semble pas jusqu'à présent pouvoir être réalisée.

Nous pensons que le formalisme de description de langages, présenté au chapitre 3, permet une définition extensible des langages : nous appellerons processus d'extensibilité la mise en oeuvre, telle qu'elle est définie au paragraphe 5.2, de cette propriété d'extensibilité dans la définition des langages. Nous étudions au paragraphe 5.4 les problèmes posés par cette mise en oeuvre.

A partir du langage de base présenté au chapitre 4, nous donnons, au paragraphe 5.3, un exemple d'extension permettant d'engendrer quelques déclareurs du langage ALGØL 68.

Nous commençons, au paragraphe 5.1, par organiser cette notion de processus d'extensibilité, dont nous avons dit le caractère récursif au chapitre 2, par la définition d'un arbre d'extensibilité.

5.1 - ARBRE D'EXTENSIBILITE -

5.1.1. - Niveau d'extensibilité -

Le processus d'extensibilité est organisé de manière récursive, de la façon suivante : à partir du langage de base, ensemble cohérent de catégories d'objets et traitements, on engendre un autre ensemble cohérent de catégories d'objets et traitements, constituant un langage dont on dit qu'il est engendré directement par extensibilité à partir du langage de base. Ce langage pouvant servir de langage de base pour une nouvelle application du processus d'extensibilité, on est dans la même situation, "décalée d'un niveau".

Définition 1 -

Le langage de base est dit langage de niveau d'extensibilité 0.

Définition 2 -

L'application du processus d'extensibilité à partir d'un langage de niveau d'extensibilité n , engendre un langage de niveau d'extensibilité $n + 1$.

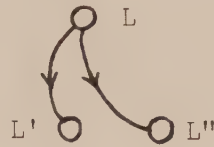
Remarque -

Deux applications différentes du processus d'extensibilité à partir d'un même langage de niveau d'extensibilité n produisent deux langages différents de même niveau d'extensibilité $n + 1$.

La génération directe d'un langage L' de niveau $n + 1$ à partir d'un langage L de niveau n peut se représenter par un graphe à deux sommets représentant les deux langages, reliés par un arc représentant l'application du processus d'extensibilité :



Deux langages différents L' et L'' engendrés directement par extensibilité à partir du même langage L peuvent se représenter par :



5.1.2 - Direction d'extensibilité -

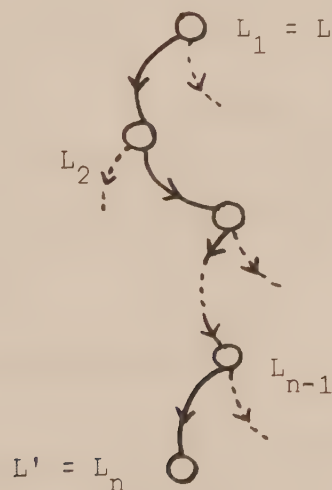
Deux langages L et L' sont dits appartenir à la même direction d'extensibilité s'il existe une suite finie $(L_i)_{i=1,n}$ de langages telle que :

$$L_1 = L$$

$$L_n = L'$$

et L_{j+1} est engendré directement par extensibilité à partir de L_j , pour tout j de 1 à $n-1$.

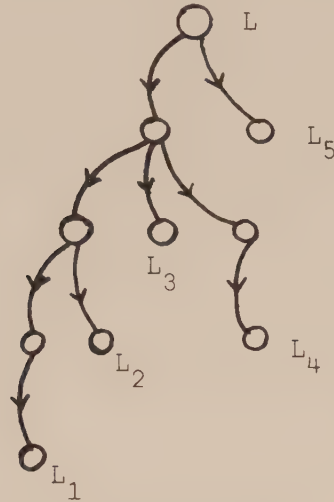
La suite $(L_i)_{i=1,n}$ définit une direction d'extensibilité, construite à partir de L_1 et menant à L_n , qu'on peut représenter par le graphe suivant :



5.1.3 - Arbre d'extensibilité -

Pour que la détermination du niveau d'extensibilité d'un langage ne soit pas ambiguë, on organise les directions d'extensibilité en arbre.

On appelle arbre d'extensibilité, de racine L, un ensemble de directions d'extensibilité construites à partir de L.



Exemple de graphe d'extensibilité de racine L, et dont L_1 , L_2 , L_3 , L_4 et L_5 sont les feuilles des cinq directions d'extensibilité différentes construites à partir de L.

L'intérêt de la construction d'un tel arbre est, pensons-nous, de permettre d'optimiser la recherche du langage minimal à partir duquel on peut engendrer une direction d'extensibilité menant à un langage donné. Cette optimisation sera naturellement essentielle quant au coût de la construction du compilateur du langage considéré.

Nous verrons au paragraphe 5.3 comment on peut construire un arbre d'extensibilité dont la racine est le langage de base défini au chapitre 4 et dont une direction d'extensibilité mènera au langage ALGØL 68.

5.2 - PROGRAMME D'EXTENSIBILITE -

La propriété d'extensibilité faisant partie de la définition formelle des langages, le processus qui est sa mise en oeuvre doit être décrit par le système de grammaires qui engendre le langage de base et défini pour tout langage engendré à partir de celui-ci, si l'on souhaite que ce nouveau langage conserve la propriété d'extensibilité.

On retiendra du procédé d'extensibilité syntaxique, par déclara-

tion de macros syntaxiques, l'idée que, sous certaines conditions, des symboles non terminaux de la grammaire engendrant un langage puissent apparaître dans des productions terminales de celle-ci, c'est-à-dire dans les phrases de ce langage. Pour ce qui concerne la grammaire conceptuelle engendrant le langage de base, l'extensibilité consiste à ce que celle-ci prévoit la possibilité de définition de nouvelles classes d'objets : pour cela les traitements du langage de base opèrent sur les concepts engendrés par la grammaire conceptuelle, et définissent de nouveaux concepts. Il faut donc que ces concepts soient des objets du langage, faisant partie de classes engendrées par la grammaire conceptuelle : ces classes sont celles engendrées, à partir de l'axiome CONCEPT, dont la dérivation des symboles non terminaux PRIMITIF ou EPRIMITIF est ~~n~~ notion.

5.2.1 - Le primitif "notion" -

Nous étudions dans ce paragraphe tous les objets de classe notion, c'est-à-dire les objets dont la classe est engendrée dans la grammaire conceptuelle du langage de base, par une dérivation incluant les symboles terminaux ~~n~~ notion.

5.2.1.a - Déclaration d'une constante de notion -

L'ensemble des propriétés caractérisant une constante de notion a été donné au paragraphe 3.2.5.

Comme pour tout genre de constante, une déclaration de constante de notion peut être implicite ou explicite suivant le type de sa désignation :

- une formule peut apparaître en partie droite d'une déclaration de constante de notion : elle est constituée d'une suite d'identificateurs de notion, et de symboles terminaux, syntaxiques ou sémantiques, du langage dans lequel est élaborée la déclaration,

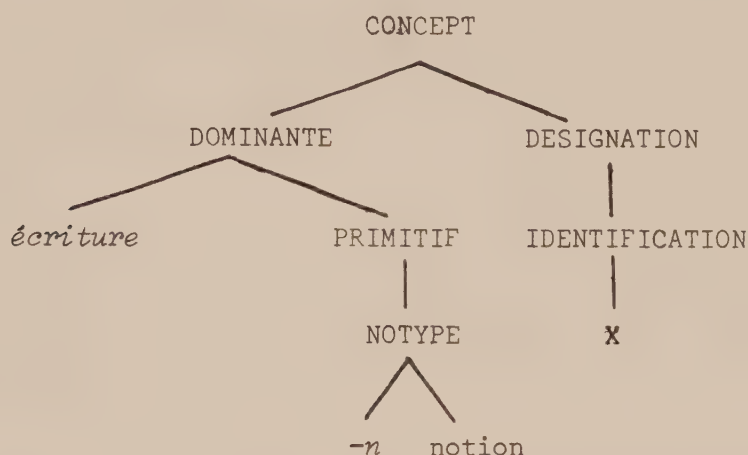
- une notation de notion est un symbole terminal de la grammaire,

- un identificateur de notion est une suite de lettres majuscules : les règles de construction, assurant notamment qu'il n'y ait pas deux occurrences de définition différentes du même identificateur, dans une partie d'un programme où ces deux définitions sont valides, sont les mêmes que pour tout autre identificateur de constante.

Exemple : notion $X = Y$ où Y est une constante de notion précédemment déclarée.

L'élaboration d'une telle déclaration consiste en sa désignation et en l'exécution d'une écriture- n sur une zone dominante, élément d'une table des identificateurs de notions.

L'objet identifié par X après élaboration de cette déclaration est de classe constante de notion, dont la dérivation dans la grammaire conceptuelle est la suivante :



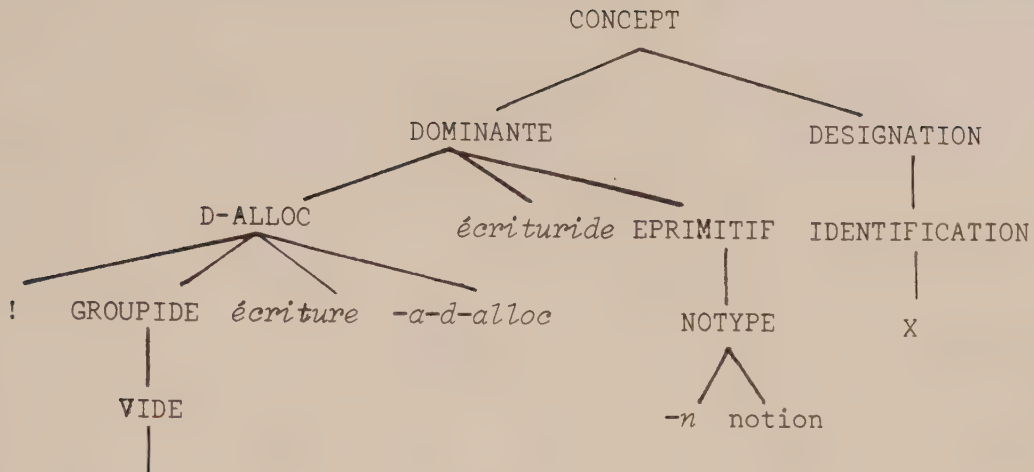
5.2.1.b - Réalisation d'une variable de notion -

L'élaboration d'une déclaration de constante d'adresse peut réaliser une variable de notion.

Exemple : ! notion $X = Y$ où Y est une variable de notion précédemment réalisée.

A une zone dominante est allouée dynamiquement une zone dominée sur laquelle peuvent être exécutées des écritures- n .

L'objet identifié par X , après élaboration de cette déclaration, est de classe constante d'adresse, dont la dérivation dans la grammaire conceptuelle est la suivante :



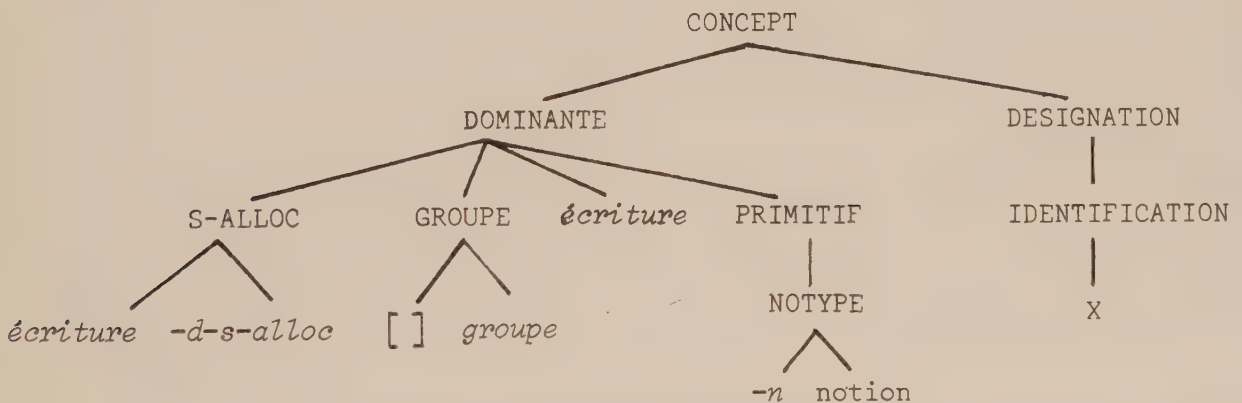
5.2.1.c - Déclaration d'une constante composée de notion -

Exemple : [] notion X = Y

où Y est une constante composée de notion, précédemment déclarée.

A une zone dominante, sur laquelle est exécutée une écriture-d est allouée statiquement une zone dominante composée, dont chaque élément est une constante de notion.

L'objet identifié par X, après élaboration de cette déclaration est de classe constante composée de notion, dont la dérivation dans la grammaire conceptuelle est la suivante :



5.2.1.d - Réalisation d'une variable composée de notion -

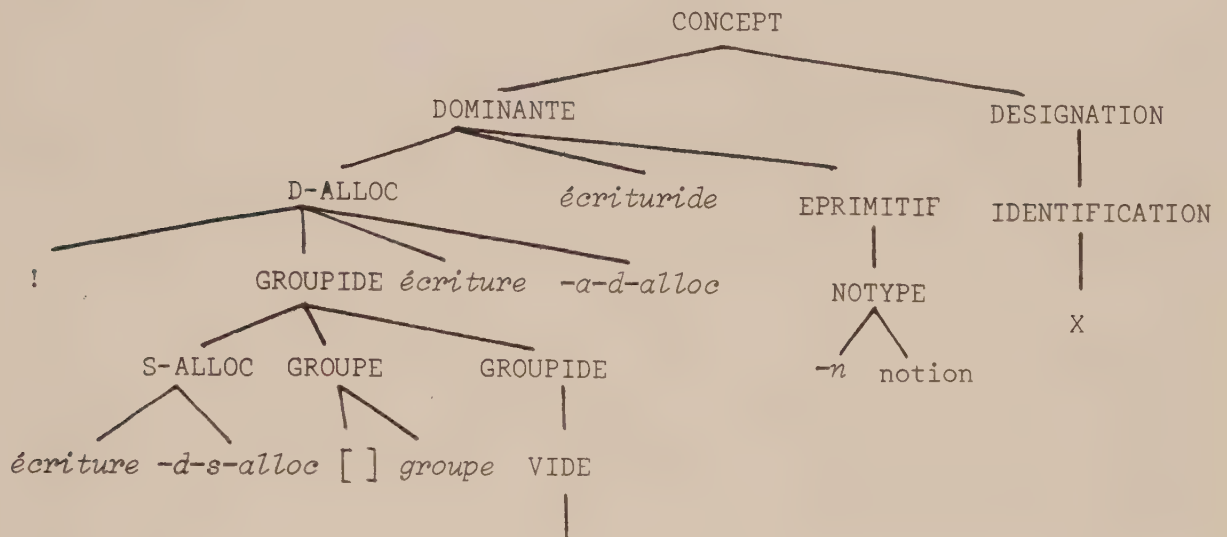
L'élaboration d'une déclaration de constante d'adresse composée peut réaliser une variable composée de notion.

Exemple : ! [] notion X = Y

où Y est une variable composée de notion, précédemment réalisée.

Chaque élément de la zone dominée composée est une variable de notion.

L'objet identifié par X, après élaboration de cette déclaration, est de classe constante d'adresse composée, dont la dérivation dans la grammaire conceptuelle est la suivante :



De même que pour les autres genres, il existe des valeurs de classe (variables multiples de)ⁿ notion, pour tout n.

5.2.2. - Programme d'extensibilité -

La réalisation du processus d'extensibilité est assurée par un ensemble de traitements exprimés dans le langage à partir duquel est définie l'extension. Ces traitements opèrent sur des concepts, qui sont des objets de classe notion, et définissent un nouveau concept, c'est-à-dire un nouvel objet de classe notion.

Ces traitements peuvent être certains algorithmes engendrés par la grammaire opérationnelle, comme la déclaration ou l'affectation. Ils peuvent également être des évaluations d'expressions ou toute instruction faisant partie des instructions du langage à partir duquel est définie l'extension.

5.2.2.a - Définition -

On appelle instruction d'extensibilité toute phrase du langage où apparaissent une ou plusieurs notions, qui sont soit des symboles non terminaux de la grammaire conceptuelle de ce langage, soit des notions précédemment déclarées.

Les notions apparaissant dans une instruction d'extensibilité tiennent lieu de toutes leurs dérivations dans la grammaire conceptuelle et des classes d'objets qu'elles engendrent, qui conservent les propriétés syntaxiques et sémantiques qui les caractérisaient dans le langage à partir duquel est définie l'extension.

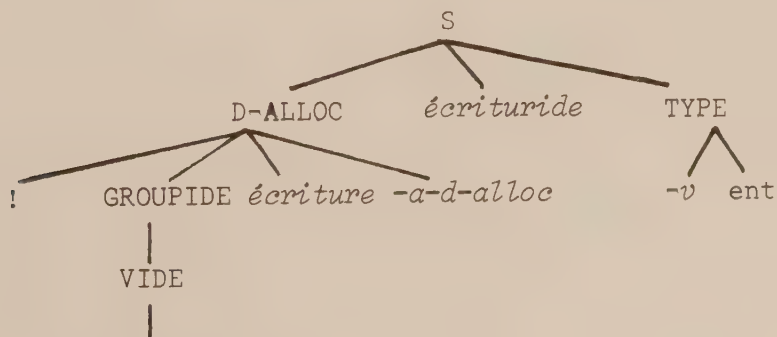
Les opérateurs apparaissant dans une instruction d'extensibilité tiennent lieu de l'algorithme engendré par la grammaire opérationnelle et dont ils sont la représentation syntaxique : ils permettent de définir une nouvelle classe d'objets, en décrivant les propriétés syntaxiques et sémantiques qui la caractérisent.

5.2.2.b - Exemple -

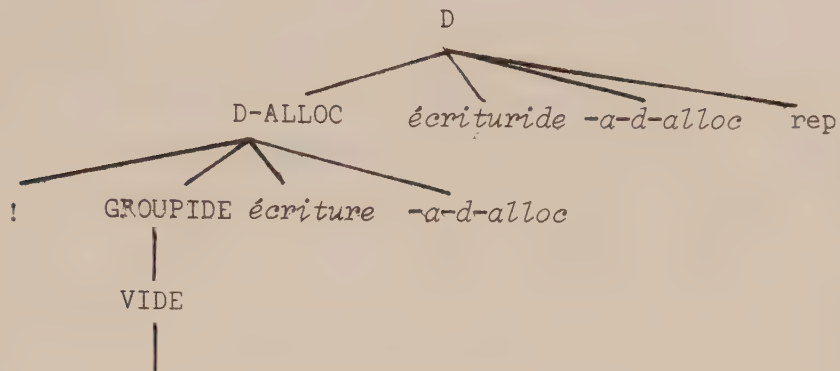
On trouve, en partie source d'une affectation à une variable de notion, énoncée en paragraphe 5.3, l'expression suivante :

D-ALLOC *écrivturide-a-d-alloc* rep:=D-ALLOC *écrivturide* TYPE

Le symbole syntaxique terminal "==" est associé à l'opérateur "affectation" qui assure la liaison entre deux actions de genre adresse : Cet opérateur a pour opérande source, la dérivation suivante :



et pour opérande destination, le noeud *écriture* de la dérivation suivante :



L'élaboration de cette expression définit, par la relation entre ces deux structures arborescentes, un mode de reconnaissance et d'accès aux valeurs de type pointeur statique, exprimés syntaxiquement par la suite de symboles terminaux !repent, équivalents aux valeurs de mode rep rep ent du langage ALGØL 68.

L'ensemble des propriétés sémantiques caractérisant cette classe de valeurs est constitué :

- d'une zone dominante sur laquelle peut être exécutée une et une seule action primitive écriture-a,
- d'une action primitive désignation de cette zone,
- d'une allocation dynamique d'une zone dominée sur laquelle peuvent être exécutées, un nombre arbitraire de fois, des écritures-a, qui allouent dynamiquement une seconde zone dominée sur laquelle peuvent être exécutées, un nombre arbitraire de fois, des écritures-v.

5.2.2.c - Programme d'extensibilité -

On appelle programme d'extensibilité de niveau n, un ensemble d'instructions d'extensibilité dont l'exécution permet d'engendrer un langage de niveau d'extensibilité n+1, à partir d'un langage de niveau n, en délivrant des moyens de reconnaissance et d'accès à un nouvel ensemble de catégories d'objets.

Nous proposons maintenant un exemple de programme d'extensibilité, de niveau 0, dont l'objet est d'engendrer certains déclareurs du langage ALGØL 68 : l'étude de cet exemple nous permettra d'aborder au paragraphe 5.4 les problèmes de compilation et d'évaluation posés par un programme d'extensibilité : ne sont pas abordés ici, les problèmes de portée d'une extension et du retour à un niveau d'extensibilité inférieur dans l'arbre d'extensibilité, qui seront résolus dans le cadre général de la structure et de l'enchaînement des instructions, que nous n'avons pas traités dans le modèle de formalisation, nous en réservant une étude approfondie pour un travail ultérieur.

5.3 - EXTENSIBILITE ALGØL 68 -

5.3.1 - Problèmes posés par cette extension -

On pourra voir sur l'exemple que nous proposons dans le paragraphe 5.3.2 que réaliser une extension n'est pas nécessairement "agrandir" le langage en l'enrichissant de nouveaux types d'objets ou traitements : le problème est en fait de créer un nouvel ensemble de catégories d'objets et traitements, à partir de ceux existants.

L'exemple cité en 5.2.2.b montre en particulier que par rapport au langage de base, à partir duquel nous proposons de faire l'extension, le langage ALGØL 68 est un langage à indication et contrôle statique de mode : cette propriété est une caractéristique de la direction d'extensibilité construite à partir du langage de base et devant mener au langage ALGØL 68.

Un des principaux objets de l'extension est donc de construire à partir des catégories de valeurs du type variable d'adresse, simple ou multiple, des catégories de valeurs du type pointeur statique : nous avons vu dans l'exemple 5.2.2.b comment est réalisée cette construction.

Un second problème posé par l'extension est l'utilisation des valeurs de type variable de descripteur, pour réaliser les tableaux flexibles ALGØL 68.

Enfin l'élimination de certaines valeurs, comme les valeurs formelles, est réalisée, on le verra au paragraphe 5.3.3, par un choix convena-

ble d'axiome dans la nouvelle grammaire définie.

5.3.2 - Programme d'extensibilité -

5.3.2.a - Remarques préliminaires -

Le programme d'extensibilité proposé est de niveau 0, c'est-à-dire qu'il est écrit dans le langage de base défini au chapitre 4.

A partir de symboles non terminaux de la grammaire conceptuelle du langage de base et des opérateurs "déclaration" et "affectation" de ce même langage, le programme d'extensibilité engendre les déclareurs ALGØL 68 de la forme :

$[]^* \{ [], \text{rep}, \text{flex} \}^* \text{TYPE}$
ou $[]^* \text{TYPE}$

exprimés dans la représentation canonique, c'est-à-dire les déclareurs de la forme :

$[]^* \underline{\text{rep}} \{ [], \underline{\text{rep}}, \underline{\text{flex}} [] \}^* \text{TYPE}$
ou $[]^* \text{TYPE}$

exprimés dans la représentation usuelle ALGØL 68.

5.3.2.b - Programme d'extensibilité -

! notion VALEUR = tas ; ! notion VALGØL = tas ;

VALEUR := (D-ALLOC *écriture* TYPE)
ou (D-ALLOC *écriture-a-d-alloc* rep:=VALEUR)
ou (D-ALLOC *écriture-d-s-alloc* flex
groupe écriture-a-d-alloc = VALEUR) ;

VALGØL := (GROUPIDE VALEUR DESIGNATION) ou
(GROUPIDE *écriture* TYPE DESIGNATION) ;

5.3.2.c - Vérifications informelles -

Ce programme contient deux déclarations de constantes d'adresses réalisant deux variables de notion, identifiées respectivement par VALEUR et VALGØL.

L'affectation à la variable VALEUR est récursive dans deux des éventualités de la formule logique qui en constitue la source.

Elle permet d'engendrer :

1/ par application de la première éventualité, les déclareurs de la forme $! []^* \text{TYPE}$,

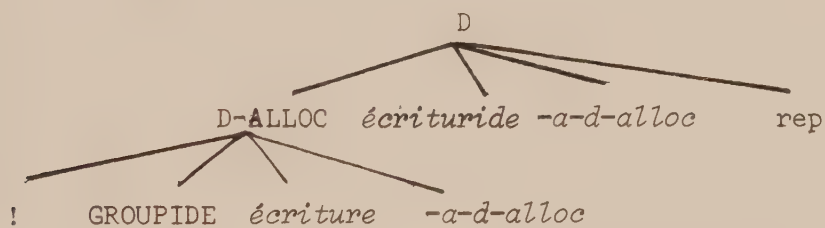
2/ par application de la deuxième éventualité :

. élaborée une fois, les déclareurs de la forme $! []^* \text{rep} []^* \text{TYPE}$

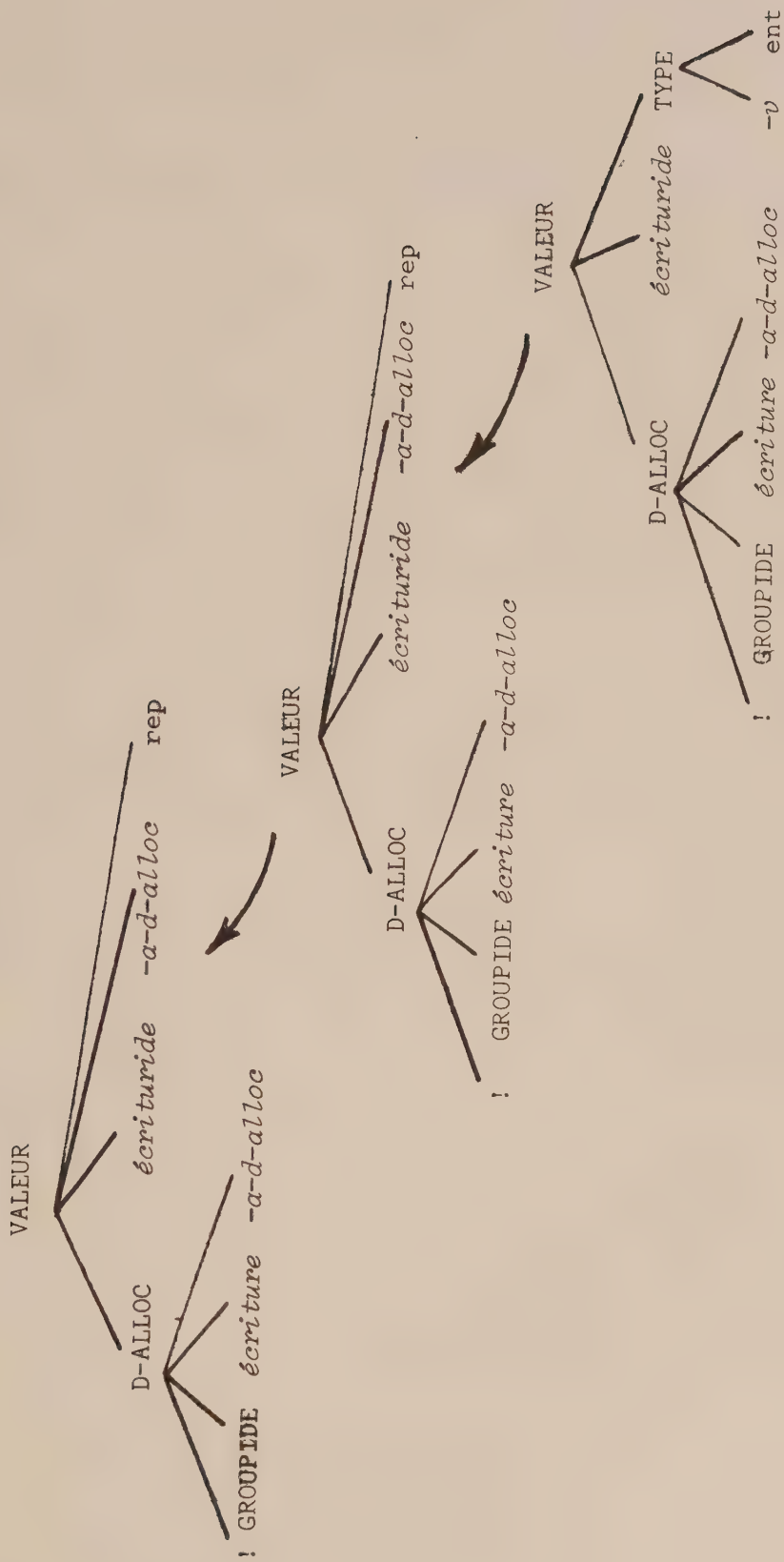
L'identificateur VALEUR désigne alors une variable de notion dont la valeur est associée à une classe d'objets, dont les propriétés sont énoncées en 5.2.2.b,

. élaborée un nombre arbitraire de fois, les déclareurs de la forme $! \{ [], \text{rep} \}^* \text{TYPE}$, où le symbole terminal rep apparaît au moins une fois.

L'objet de classe notion, désigné par VALEUR est alors opérande source de l'opérateur "affectation", dont l'opérande destination est le noeud *écriture* de la dérivation.



Après l'élaboration de cette affectation la valeur de la variable de notion VALEUR est associée à une classe d'objets dont les propriétés sont, par exemple, exprimées par ces trois dérivation dans la grammaire conceptuelle liées par l'action de l'opérateur "affectation", schématisé par ↩ :



3/ par application de la troisième éventualité :

. élaborée une seule fois, les déclareurs de la forme

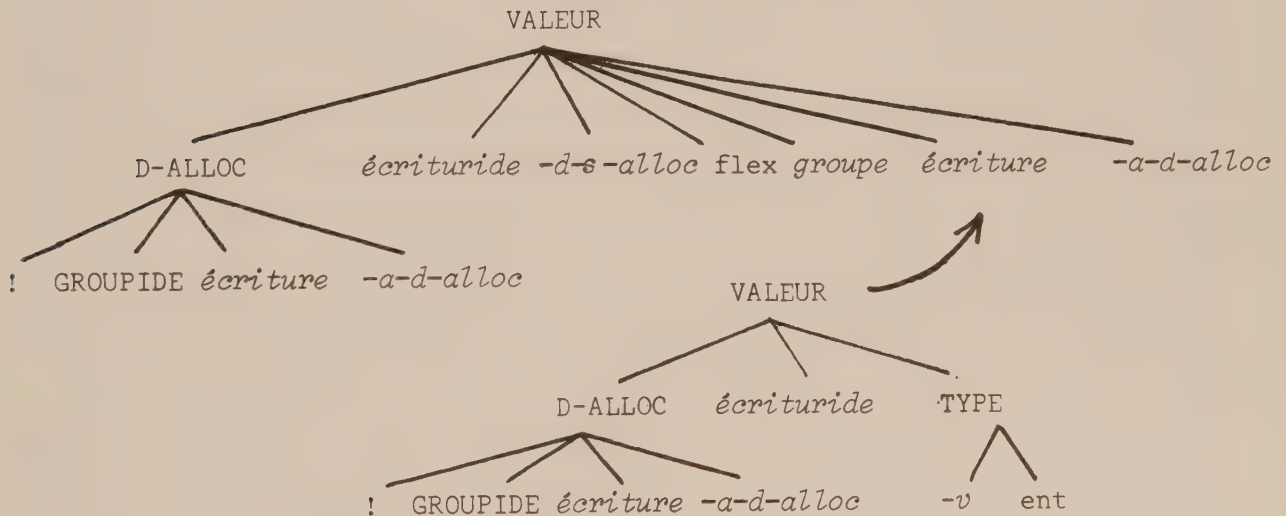
! []* flex []* TYPE

. élaborée un nombre arbitraire de fois, les déclareurs de la forme

! { [], flex }* TYPE, où le symbole terminal flex apparaît au moins une fois.

Chaque élément de la zone composée définie par la partie gauche de la déclaration qui constitue cette éventualité est un objet de classe constante d'adresse : l'action primitive écriture-a est exécutée dans l'algorithme de l'opérateur "déclaration" qui assure la liaison avec l'action primitive de lecture de même genre qui apparaît dans la dérivation de la valeur de la variable de notion VALEUR. Le symbole sémantique *groupe* précisera que l'exécution de la dernière occurrence d'écriture apparaissant dans l'ordre lexicographique est répétée.

Le résultat de l'élaboration de cette déclaration, peut se représenter, par exemple, par cet ensemble de dérivations dans la grammaire conceptuelle, liées par une représentation schématique de l'opérateur "déclaration":



4/ par application de la première éventualité ou d'un nombre arbitraire de fois de la deuxième ou troisième éventualité, les déclarateurs de la forme `!{ [ ], rep, flex}* TYPE`

L'affectation à la variable de notation VALGØL permet d'engendrer tous les déclarateurs spécifiés au paragraphe 5.3.2.a.

5.3.3 - Le nouvel axiome de la grammaire -

L'exécution d'un tel programme d'extensibilité a pour résultat la définition d'un nouvel ensemble de règles de grammaire : pour réaliser convenablement la définition formelle du langage ainsi engendré, il convient de préciser l'éventuel nouvel axiome de sa grammaire : on remarque en effet que si CONCEPT reste, dans le cas considéré l'axiome, on engendre d'autres déclarateurs que ceux indiqués, puisqu'on engendre en outre tous ceux du langage de base.

Pour éliminer une ambiguïté de définition on adopte les deux règles suivantes :

Règle_1 -

Il existe parmi les notions (constantes ou variables) déclarées dans un programme d'extensibilité une et une seule notion apparaissant en partie gauche d'une déclaration ou d'une affectation et n'apparaissant pas en occurrence d'utilisation, sauf éventuellement dans la partie droite de sa propre déclaration ou en partie source d'une affectation dont elle est la destination.

Règle_2 -

L'axiome de la grammaire définie par un programme d'extensibilité est l'unique notion, qui dérive de la notion tenant lieu de 'AXIOME', dans la définition du langage dans lequel est écrit le programme d'extensibilité, par l'une des dérivations :

'AXIOME' * *écriture-n* notion DESIGNATION
ou 'AXIOME' * D-ALLOC *écriture-n* notion DESIGNATION,
et qui satisfait aux conditions de la règle 1.

La seconde variable de notion du programme d'extensibilité ci-dessus déclarée par :

! notion VALGØL = tas,
apparaissant dans la seule instruction d'affectation :

VALGØL := (GROUPE VALEUR DESIGNATION) ou (GROUPE *écriture*
TYPE DESIGNATION)

est, d'après les règles 1 et 2, l'axiome de la grammaire ainsi définie.

5.4. - MISE EN OEUVRE DE L'EXTENSIBILITE -

Dans les exemples de mécanismes d'extensibilité proposés, où interviennent les notions de la grammaire qui engendre le langage de base, principalement les extensions par définition de macros syntaxiques, ces notions sont seulement considérées comme éléments non terminaux de cette grammaire.

Lorsque se posent des questions de définition, d'évaluation ou d'utilisation de ces macros syntaxiques, on se heurte à différents problèmes. Ainsi, si l'on veut accroître la fiabilité et l'efficacité du mécanisme, doit-on envisager pour une définition de macro :

- d'une part une analyse syntaxique, qui assure en particulier que la phrase analysée est exprimée dans le langage à partir duquel on définit l'extensibilité,

- d'autre part une évaluation de cette définition, par une élaboration partielle de sa sémantique, par exemple par la construction d'un arbre abstrait.

C'est-à-dire qu'il faut concevoir pour ces objets particuliers que sont les phrases du langage où sont "tolérés" des symboles non terminaux les mêmes traitements que pour les phrases, au sens classique du langage.

Pour ce qui concerne l'analyse syntaxique, on peut, moyennant certaines contraintes, comme nous l'avons vu au chapitre 2, obtenir un arbre syntaxique, dont certaines feuilles correspondent à des symboles non terminaux de la grammaire et que l'on considère, pour l'utilisation, comme des paramètres formels.

Pour l'évaluation d'une définition de macro **syntaxique**, on construit un programme abstrait représenté comme un arbre, dont certaines feuilles

sont laissées indéterminées, qui correspondent aux paramètres formels de l'extension. Pour une utilisation, on assemble les programmes abstraits des paramètres réels, au précédent, au moyen d'opérateurs sur les structures arborescentes.

Dans le processus que nous proposons, les notions apparaissant dans les instructions d'extensibilité, ne sont pas considérées comme des symboles non terminaux de la grammaire, mais comme des identificateurs qui désignent des objets associés à des productions terminales de la grammaire : ces objets tiennent lieu de toutes les dérivations du symbole non terminal considéré.

Dès lors les problèmes d'analyse syntaxique et d'évaluation sont les problèmes généraux d'analyse syntaxique et d'évaluation des phrases du langage : en particulier, nous avons remarqué dans le paragraphe précédent que l'évaluation des instructions d'extensibilité consiste en l'expression des algorithmes associés aux opérateurs qui y apparaissent, sous la forme d'un ensemble de relations entre des structures arborescentes.

Dans un but d'efficacité, nous concevons que l'exécution d'un programme d'extensibilité doit précéder toute utilisation du langage engendré par l'extension qu'il produit. Cependant, il n'est pas exclus de penser que des instructions ou des procédures d'extensibilité pourront être insérées dans un programme quelconque : il faudrait alors envisager une compilation de ce programme en deux temps, le premier considérant les instructions d'extensibilité.

Un tel procédé devrait naturellement être réservé pour une utilisation très spécifique d'une certaine extension.

CONCLUSION -

Le processus d'extensibilité proposé est la mise en oeuvre d'une définition extensible des langages : cette extensibilité est plus qu'intégrée à la définition des langages, elle est leur mode de définition.

Alors que les mécanismes classiques proposent une possibilité de réalisation de certaines extensions par l'utilisation de traitements particuliers, la définition extensible proposée sous la forme des grammaires conceptuelle et opérationnelle, assure, à partir d'un certain ensemble d'objets et de traitements sur ces objets, la définition d'un autre ensemble d'objets et de traitements sur ces nouveaux objets.

En considérant que les traitements sont définis sur tous les objets dont la classe est exprimée par une production terminale de la grammaire conceptuelle, et en définissant par cette grammaire une classe d'objets associée aux notions, on réalise l'aspect de l'extensibilité qui consiste en la définition et la reconnaissance d'un nouvel ensemble d'objets.

En considérant tout opérateur, comme production terminale de la grammaire opérationnelle, la dérivation qui l'a engendré définit son action sur tous les objets accessibles dans le langage, en particulier sur ceux dont la classe est décrite par une dérivation dans la grammaire conceptuelle à partir d'une notion définie par extension : on réalise ainsi cet aspect de l'extensibilité, appelé extension de la sémantique dynamique, dans les méthodes de description interprétatives, qui consiste à définir un nouveau répertoire d'opérateurs pour l'interpréteur.

La récursivité de la définition de l'extensibilité pose le problème de l'initialisation du processus par le choix du langage de base. Celui-ci, tel qu'il a été défini au chapitre 4 donne accès à un certain ensemble de catégories d'objets et de traitements qui nous a paru, pour ce travail, être le plus apte à décrire correctement et simplement le processus d'extensibilité.

Il va de soi que le langage de base proposé n'est pas le noyau minimal, s'il existe, pouvant servir de racine à un arbre "universel" d'extensibilité.

Nous le considérons comme un noeud, satisfaisant pour l'étude proposée, d'un tel arbre d'extensibilité : les catégories d'objets et traitements que nous avons définis, à priori, doivent donc être considérés comme des résultats d'extensions antérieures, à partir d'un langage de base qui reste à définir : chaque notion de la grammaire conceptuelle est donc le résultat de l'élaboration d'une déclaration de constante de notion ou de la réalisation d'une variable de notion. En particulier l'axiome 'CONCEPT' doit être considéré comme satisfaisant aux règles 1 et 2 du paragraphe 5.3.3, et dériver d'une notion tenant lieu de 'AXIOME' dans un langage de niveau d'extensibilité inférieur.

Un des buts d'une étude ultérieure sera naturellement de tendre vers un langage de base minimal : pour tendre vers celui-ci on pourra étudier une direction d'extensibilité formée de sous-langages du langage de base proposé au chapitre 4, suivant un arbre d'extensibilité de la forme :



CONCLUSION

Que le lecteur comprenne bien que ce qui vient d'être exposé se veut être un modèle de description : s'y cotoient et s'entremêlent énoncés d'intentions dans les principes retenus pour la formalisation, dont on tente de montrer la mise en oeuvre pour certains aspects, et exposés précis d'un ensemble de concepts : objets ou actions, nécessaires pour celle-ci.

C'est que nous avons essayé, tout en voulant présenter le plus correctement possible le langage engendré, c'est-à-dire le fruit qui résultera de sa définition formelle, de ne pas en faire perdre de vue la genèse, c'est-à-dire le principe général de l'organisation du modèle de description.

Nous ne présentons, en fait, que le premier pas d'une étude dont la démarche originale en comportera beaucoup d'autres : il fallait donc que le premier pas fût solide et que la démarche ainsi engagée fût assez largement entrevue. Nous espérons n'avoir pas trop failli à cette double exigence.

Ceci quant à la forme.

Quant au fond, on peut, peut-être artificiellement, partager en trois catégories de problèmes, les prochaines études, pour certaines déjà en cours, devant réaliser cette démarche ultérieure :

- le problème de la définition formelle du langage de base,
- le problème de la construction d'un arbre d'extensibilité,
- le problème de l'implémentation,

l'ordre dans lequel nous les citons n'ayant sans doute que très peu de rapport avec l'ordre chronologique dans lequel ils seront traités au cours de travaux à venir.

1 - Le problème de la définition du langage de base se pose à deux niveaux.

Nous nous sommes attachés à présenter ici l'aspect grammair conceptuelle du modèle de description, dont nous n'avons que très sommairement présenté les autres aspects, en en illustrant seulement par des exemples,

informels pour la plupart, les diverses relations. Ce qui concerne la grammaire opérationnelle et la génération par la grammaire conceptuelle de classes d'objets autres que les valeurs devra être traité formellement, ainsi que l'étude de l'organisation générale du modèle.

Dans un second temps, nous devons transformer le modèle de description en une définition formelle du langage de base, incluant d'autres concepts que ceux cités pour exemples : alors seulement sera engendré un "vrai" langage.

2 - Le problème de la construction d'un arbre d'extensibilité est lui aussi à plusieurs directions : la première est la définition systématique des directions d'extensibilité menant aux langages évolués classiques. Elle passe, comme nous le disons dans la conclusion du chapitre 5, par l'évolution de la racine de l'arbre d'extensibilité, jusqu'à la recherche d'une racine "idéale". Enfin une autre étude à mener est de montrer l'intérêt que peut présenter la génération par extensibilité de tel ou tel langage spécifique, adapté à tel ou tel problème particulier.

3 - Enfin, il va de soi qu'il faut assurer la mise en oeuvre pratique de ce formalisme, en en concevant une implémentation : le lecteur aura remarqué que, tant par la conception même de ce formalisme, que par la présentation, aussi souvent que possible, de schémas sémantiques, suggérant un modèle d'implémentation mémoire, qui reste à optimiser, le souci de "réalisme" fut une des préoccupations majeures qui ont guidé ce travail.

BIBLIOGRAPHIE

- [An 71] J. ANDRE -
Introduction à SIMULA 67.
Séminaire de programmation de l'Université de Toulouse,
Janvier 1971.
- [Ar G 74] B. ARLETTAZ et J.C. GIRARD -
Définition de la syntaxe et de la sémantique des langages de
programmation par le langage de Vienne.
Séminaire de l'Institut de Programmation, Paris, Mai 1974.
- [Bo 64] J.C. BOUSSARD -
Etude et réalisation d'un compilateur ALGØL 60 sur calculatrice
électronique du type IBM 7090/94 et 7040/44.
Thèse d'état, Grenoble, 1964.
- [Br 71] B.M. BROSGOL -
An Implementation of ECL Data Types.
Proceedings of the International Symposium on Extensible
languages. SIGPLAN Notices. Vol 6, n° 12, Décembre 1971.
- [Mc 66] J. Mc CARTHY -
A formal description of a subset of Algol.
Formal language description languages for Computer Programming.
North-Holland Publishing Company. Amsterdam, 1966.
- [Ch F 68] T.E. CHEATHAM J.R., A. FISCHER, P. JORRAND -
On a basis for ELF. An extensible language facility.
Fall Joint Computer Conference, 1968.
- [Co 74] Cours de l'Ecole d'Eté - Aspects de la théorie de la Programmation
Ecole d'Eté d'informatique. Tarbes, 1974.

- [Cu D 75] CUNIN P.Y., DELAUNAY M., SIMONET M., VOIRON J. -
Compilateur ALGØL 68 sur IBM 360.
Exposé au séminaire de l'IRIA, Paris, Avril 1975.
- [Da M 66] D.J. DAHL, B. MYHRHAUG, K. NYGAARD -
Simula 67 Common Base.
Communication ACM, Septembre 1966.
- [De R 74] G. DEMARS, J.C. RAULT, G. RUGGIU -
Le langage et les systèmes APL.
Masson, 1974.
- [Di 72] E.W. DIJKSTRA -
On the necessity of correctness proofs.
Advanced course on computer systems architecture. Alpe d'Huez,
Décembre 1972.
- [Gi 66] S. GINSBURG -
The mathematical theory of context-free languages.
Mc Graw-Hill Book Company, New-York, 1966.
- [Gr 73] M. GRIFFITHS -
Relationship between definition and implementation.
Advanced course in software engineering.
Lecture Notes in Economics and Maths Systems. Springer
Verlag, 1973.
- [Gr A 72] GROUPE ALGØL DE L'AFCEC -
Définition du langage algorithmique ALGØL 68. Hermann, 1972.
- [Gr A 75] GROUPE ALGØL DE L'AFCEC -
Le langage algorithmique ALGØL 68. Hermann, 1975.
- [Ha 71] M. HAMMER -
An alternative approach to macro-processing.
Proceedings of the international symposium on extensibles
languages. SIGPLAN Notice. Vol 6, n° 12, Décembre 1971.

- [Ho 69] C.A.R. HOARE -
"An axiomatic basis for computer programming".
Comm. ACM 12, pp. 576-581, Octobre 1969.
- [Ho W 72] C.A.R. HOARE et N. WIRTH -
An axiomatic definition of the programming language
PASCAL. Berichte der fach gruppe computer wissenschaftenn°6, 1972.
- [Ic 71] J.C. ICHBIAH -
Extensibility in simula 67.
Proceedings of the international symposium on Extensible
languages. SIGPLAN Notice. Vol 6, n° 12, Décembre 1971.
- [Jo B 72] P. JORRAND, D. BERT -
On some basic concepts for extensible programming languages.
Proceedings of international computing symposium.
CINI Foundation. Venise, Avril 1972.
- [Jo 75] P. JORRAND -
Contribution au développement des langages extensibles.
Thèse d'état, Grenoble, 1975.
- [Kn 71] D.E. KNUTH -
Examples of Formal Semantics.
Lecture notes in Math., n° 188, Springer Verlag, 1971.
- [Le 66] LEAVEN WORTH -
Syntax macros and extended translation.
Communications of the ACM. Novembre 1966, Vol. 9, n° 11.
- [Le 72] LEROY H.
Pointeur à des constantes.
Proposition de révision au rapport de définition du langage
ALGØL 68, Vienne, Novembre 1972.
Egalement dans ALGØL BULLETIN, N° 50.

- [Lo B 73] B. LORHO, L. BLAIZOT -
Projet DELTA. Système de description de langages et de traducteurs par attributs.
IRIA, rapport n° 20, 1973.
- [Lo 75] B. LORHO -
Compilation par attributs sémantiques.
Exposé à l'Institut de programmation, Paris, Février 1975.
- [Lo 74] B. LORHO -
De la définition à la traduction des langages de programmation : méthode des attributs sémantiques.
Thèse, Université de Toulouse, 1974.
- [Lu W 69] P. LUCAS, K. WALK -
On the formal description of PL/1.
IBM Laboratory, Vienna.
Annal review in automatic programming. Vol 6, part. 3, 1969.
- [Ne 71] E.J. NEUHOLD -
The formal description of programming languages.
IBM systems Journal n° 2, 1971.
- [Pa 73] C. PAIR -
Calculs, ensemble de calculs, équivalence de programmes.
Convegno Informatica Teorica, Rome, 1973.
- [Sa R 73] D. SAMSON, L.P.A. ROBICHAUD -
Execute APL/360.
Université Laval - QUEBEC - P.Q. Canada, Juillet 1973.
- [Sc S 71] D. SCOTT, C. STRACHEY -
Towards a mathematical semantics for computer languages.
Symposium on computers and Automata. Brooklyn, 1971.
- [Sc J 70] S.A. SCHUMAN, P. JORRAND -
Définition méchanisms in extensible programming languages.
AFIPS Conference Proceedings, FJCC, 1970.

- [Vw M 73] A. Van WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK,
Ch. A. KOSTER, M. SINTZOFF, Ch. LINDSEY -
Draft revised report on the algorithmic language ALGØL 68.
Copyright, IFIP, Juillet 1973.
- [Vi 74] J. VIDART -
Extensions syntaxiques dans un contexte LL(1).
Thèse 3ème cycle, Grenoble, septembre 1974.
- [We 71] B. WEGBREIT -
The ECL programming system.
Proceedings FJJC, Vol 39, 1971.

